Università di Pisa
Scuola di Ingegneria

Corso di Laurea Magistrale in
Ingegneria Informatica per la Gestione d'Azienda

Analisi e Realizzazione di modelli di comunicazione sicura in architetture SOA

Tesi di Laurea di

Antonio Pizzurro
5 Giugno 2014

Relatori:

Prof. Gianluca Dini
Prof. Cinzia Bernardeschi
Dott. Emanuele Pardini

Anno Accademico 2013-2014

Alla mia famiglia

Abstract

Il seguente lavoro di tesi è inquadrato all'interno di un Progetto finanziato dalla Regione Toscana, chiamato PITAGORA, a cui hanno partecipato aziende private e Università; è stato svolto presso la sede fiorentina di Thales Italia in quanto Group Leader dell'intero progetto.

L'obiettivo di PITAGORA è la realizzazione di una Piattaforma Integrata per la Gestione delle Operazioni Aeroportuali e si avvale di un'infrastruttura a servizi secondo il paradigma SOA. La Piattaforma è composta da una serie di applicazioni sviluppate dai partner che vengono agganciate ad un Enterprise Service Bus (*ESB*) per mezzo di integrazione basata su *Web Service*.

L'apporto fornito da UNIPI si è focalizzato sulla messa in sicurezza delle comunicazioni tra le componenti applicative dei partner e l'ESB, con l'obiettivo di garantire requisiti quali *confidenzialità*, *integrità* e *autenticità* delle informazioni in transito su un canale di comunicazione.

Inizialmente, sono state individuate due possibili soluzioni basate sullo stato dell'arte per infrastrutture a servizi: *WS-Security* come specifica di sicurezza legata ai servizi web e il protocollo *SSL/TLS* come protocollo sicuro di trasporto; quest'ultima è risultata più idonea allo scopo.

L'adozione del protocollo SSL in mutua autenticazione ha portato all'installazione di una Certification Authority (CA), interna al dominio della Piattaforma e sotto il controllo del Group Leader, così che esso possa rilasciare credenziali, sottoforma di certificati, solo ad entità autorizzate a comunicare con la Piattaforma.

E' stato poi realizzato un prototipo di servizio web basato su tecnologia Java e installato sull'ESB e due *connettori* (in Java e C++), cioè applicazioni consumer di servizi, che assumessero il ruolo di applicazioni partner che fanno richieste presso la Piattaforma, ovvero l'ESB.

E' stata formalizzata la procedura di rilascio e gestione dei certificati e rilasciato un certificato ad ogni componente(ESB e connettori). Con l'uso dei certificati il servizio è in grado di riconoscere, oltre alla funzionalità richiesta anche l'applicazione che lo invoca e può adottare quindi politiche di accesso personalizzate; inoltre, è possibile autorizzare o meno un'applicazione ad accedere al servizio, utilizzando il concetto delle revoche dei certificati.

Sono state quindi apportate le dovute configurazioni all'ESB e ai connettori per un corretto uso del protocollo e infine, sono state effettuate delle misure di perfomance, per capire quanto l'introduzione della soluzione potesse intaccare le prestazioni del sistema.

Indice

Elenco delle figure

Introduzione

La continua richiesta di soluzioni innovative e l'incessante mutamento dei mercati e degli assets economici ci impongono sfide sempre più impegnative. Per restare al passo e non essere sommersi dall'onda del progresso e del cambiamento dobbiamo quindi essere rapidi e soprattutto flessibili se vogliamo sopravvivere. E' quasi una rinascita del Darwinismo:

> *Non è il più forte della specie a sopravvivere, né il più intelligente, ma quello che meglio sa adattarsi al cambiamento.*

Essendo l'Information Technology ormai di fatto un fattore chiave per la gestione dei processi aziendali, capita sempre più spesso che le aziende si trovino a gestire i propri processi di business per mezzo sistemi distribuiti di grande portata e a componenti eterogenee, con la conseguenza inevitabile dell'aumento della complessità generale. Abbandonando di fatto l'idea che sistemi centralizzati siano la soluzione al problema ci si è orientati verso un concetto di aggregazione, o meglio integrazione di tali sistemi individuali, in unico sistema distribuito che sia facilmente manutenibile, scalabile e a componenti interoperabili e interscambiabili. L'infrastruttura necessaria a svolgere questo ruolo è quella rappresentata dal paradigma SOA che sarà

caratterizzante per gli scopi di questo elaborato. Il seguente lavoro di tesi si inserisce in un progetto della Regione TOSCANA - Progetto PITAGORA - volto alla realizzazione di una Piattaforma software per la gestione aeroportuale. In particolare, lo studio è stato centrato su aspetti relativi alla sicurezza nelle comunicazioni tra le componenti della Piattaforma, fornendo preliminarmente un'analisi di possibili soluzioni, ricercate tra gli standard attualmente in uso, e successivamente, sulla base di una di queste soluzioni è stato modellato un prototipo di servizio con lo scopo di dimostrare l'efficacia e l'efficienza della soluzione attuata. Il testo che segue, nello specifico sarà così strutturato: nel **primo capitolo** si farà riferimento al Progetto PITAGORA, ai suoi obiettivi e risultati attesi e si darà una panoramica della tipologia di infrastruttura su cui la piattaforma si basa; nel **secondo capitolo** si fornisce una breve spiegazione delle varie tecnologie affrontate nel corso del lavoro. Il **terzo capitolo** è incentrato inizialmente sulla descrizione dei requisiti di sicurezza da garantire in termini di comunicazione, su una breve panoramica di alcuni concetti di sicurezza relativi ad algoritmi simmetrici e asimmetrici ed infine si descriveranno due possibili soluzioni applicabili per gli scopi della piattaforma; nel **quarto capitolo** si espliciteranno le motivazioni che hanno condotto verso una determinata soluzione di sicurezza e nel **quinto capitolo** si illustrerà la realizzazione di un prototipo di servizio web basato sulle stesse tecnologie di PITAGORA al quale sarà applicata la soluzione prescelta. Infine il **sesto capitolo** consiste in un'analisi in termini di performance per capire quanto l'uso della soluzione possa degradare le prestazioni della piattaforma.

Capitolo 1

Fare Integrazione: Il Progetto Pitagora

Il *Progetto PITAGORA* è un progetto di integrazione in ambiente aeroportuale volto a favorire e migliorare tutti i processi che ruotano intorno alla gestione di un'infrastruttura critica. Tramite una serie di partnership, con privati e università, il progetto mira a riunificare un insieme di servizi realizzati dai partner in modo da ottenere una piattaforma unica e coerente. L'intera infrastruttura è realizzata sulla base di *un'architettura a servizi secondo il paradigma SOA*, e ciò per favorire la comunicazione tra le parti, l'interoperabilità e il superamento delle eterogeneità tra le varie componenti che formeranno la struttura finale della piattaforma.

1.1 Service Oriented Architecture

1.1.1 Cosa significa SOA?

SOA è l'acronimo di Service oriented architecture, un modello architetturale per la creazione di sistemi, residenti su una rete, che focalizza l'attenzione sul concetto di *servizio*. E' un evoluzione dei sistemi distribuiti, nata principalmente per rispondere all'esigenza di integrare tra loro sistemi, generalmente eterogenei; in modo più astratto la SOA può essere considerata come un'architettura il cui scopo è il raggiungimento di un disaccoppiamento tra agenti software interagenti tra loro.[1] E' importante però capire che SOA rappresenta un paradigma, un concetto, una filosofia e non, come erroneamente si crede, qualcosa di reale e immediatamente fruibile. E' appunto uno stile da seguire, un modo di pensare che fornisce una guida per progettare un'architettura reale.

Da un punto di vista più formale, secondo la definizione del gruppo OASIS (Organization for the Advancement of Structured Information Standards):

> SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations[2]

Si tratta dunque di un paradigma per organizzare ed utilizzare capacità (o funzionalità) distribuite che possono essere sotto il controllo di diversi owners. Fornisce una modalità uniforme per offrire, trovare, interagire e usare tali

funzionalità per produrre gli effetti desiderati, consistenti con le aspettative previste.

In linea di principio, un sistema costruito secondo tale paradigma, è costituito da applicazioni chiamate appunto *servizi*, ben definite ed indipendenti l'una dall'altra, che risiedono su più computer all'interno di una rete. Ogni servizio mette a disposizione una certa funzionalità e può utilizzare quelle che gli altri servizi hanno reso disponibili, realizzando, in questo modo, applicazioni di maggiore complessità.

La filosofia alla base è quella di decentrare la business logic in unità logiche più piccole, per lo più indipendenti, in grado di assolvere a specifici compiti, messi a disposizione sotto forma di servizi. Questi ultimi, infatti, rappresentano il punto di forza di tutta l'architettura, in quanto permettono l'interoperabilità tra sistemi eterogenei, al fine di realizzare quello che è il cosiddetto processo di business.

1.1.2 I servizi

Un servizio può essere definito come un'entità software che fornisce funzionalità a chi lo utilizza, attraverso lo scambio di messaggi. Lo stesso gruppo OASIS ne da la seguente definizione:

> A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by service description

Dalla definizione si deduce che un servizio prevede un'*interfaccia* ed una *descrizione* di se stesso; essendo necessaria l'indipendenza tra i servizi, essi

devono fornire descrizione ed interfaccia secondo regole standard (generalmente espresse tramite XML), al fine di accedere ed essere accessibili da altri servizi. Più in generale, un servizio dovrà:

- Essere ricercabile e recuperabile dinamicamente: un servizio deve poter essere ricercato in base alla sua interfaccia e richiamato a tempo di esecuzione. Avendo definito il servizio per mezzo della sua interfaccia, ciò rende quest'ultima indipendente dalla specifica implementazione del servizio stesso.

- Essere auto contenuto e modulare: ogni servizio deve essere ben definito, completo ed indipendente dal contesto o dallo stato di altri servizi.

- Essere definito da un'interfaccia ed indipendente dall'implementazione: deve cioè essere definito in termini di ciò che fa, astraendo dai metodi e dalle tecnologie utilizzate per implementarlo. Ciò determina l'indipendenza del servizio sia dal linguaggio di programmazione che dalla piattaforma su cui è in esecuzione, quindi non è necessario conoscere come un servizio è realizzato ma solo quali funzionalità rende fruibili.

- Essere debolmente accoppiato con altri servizi (loosely coupled): un'architettura è debolmente accoppiata se le dipendenze fra le sue componenti sono in numero limitato. Questo rende il sistema flessibile e facilmente modificabile. Limitare al massimo il numero di modifiche da apportare ad un servizio al cambiamento di un altro, promuove il riuso dei servizi stessi e la loro evoluzione in maniera indipendente.

- Essere reso disponibile sulla rete: quindi dovrà essere localizzabile attraverso particolari mezzi e successivamente si dovranno poter recuperare le informazioni per quel servizio.

- Fornire delle funzionalità possibilmente a grana grossa (coarse-grained): non si devono scrivere servizi per compiere operazioni talmente semplici da non giustificare il costo dell'utilizzo del meccanismo SOA.

- Essere realizzato in modo tale da permetterne la composizione con altri servizi: un aspetto chiave di questa architettura, in quanto le applicazioni SOA saranno il risultato dell orchestrazione di più servizi che vanno a comporre il processo di business su cui si vuole operare.

1.1.3 Funzionamento di una SOA

L'architettura prevede la presenza di tre attori:

- Service Provider

- Service Consumer

- Service Registry

Il Service Provider è l'entità software che mette a disposizione una serie di funzionalità, quindi un servizio. Per rendere tale servizio fruibile da chi ne ha bisogno, è necessario che sia reso visibile in rete, ossia pubblicato. A tale scopo esistono i Service Registry, ai quali un provider fornisce le informazioni relative al servizio che fornisce. Il Service Registry possiede quindi le informazioni, come URL e modalità di accesso, di tutti i servizi

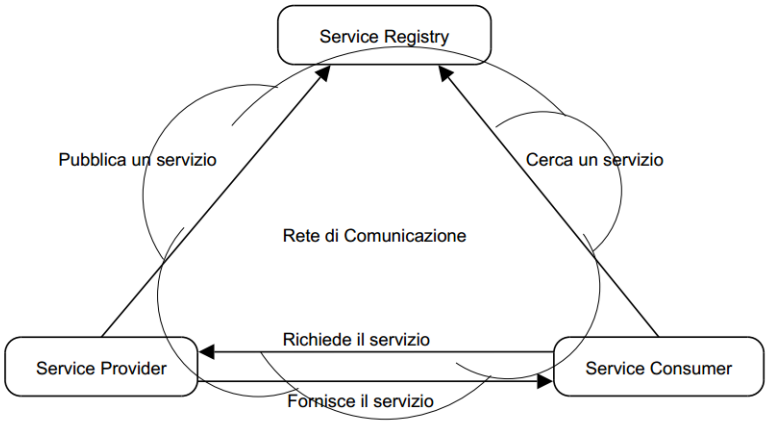


Figura 1.1: Esempio di architettura SOA

disponibili. Quando un Service Consumer vorrà utilizzare un servizio, farà richiesta al Service Registry, il quale gli fornirà tutte le informazioni ad esso relative. A questo punto il Consumer può comunicare direttamente con il Provider. Tutte queste operazioni sono svolte attraverso nell'ambito di una rete di comunicazione, che può quindi essere una rete locale o Internet.

SOA quindi si limita a definire le linee guida e le caratteristiche a cui i componenti del sistema devono attenersi al fine di poter definire quest'ultimo un'architettura a servizi. Tra le tecnologie atte a realizzare una SOA, ad oggi, quella dei Web Services (WS), di cui si parlerà in seguito, è sicuramente la più utilizzata.

1.2 Cosa vuol dire fare integrazione

In ambienti di business sempre più competitivi e dinamici l'integrazione tra applicazioni di natura eterogenea, appartenenti talvolta ad aziende diverse

e con scopi differenti, è diventata un'esigenza sempre più pressante. L'integrazione è un processo finalizzato alla possibilità scambiare informazioni tra applicazioni che utilizzano protocolli, formati dei dati e interfacce non compatibili tra di loro. Diversi approcci sono stati proposti in questi anni, ma quello su cui ci soffermeremo, su cui inoltre è improntato questo lavoro di tesi è l'approccio SOA tramite l'uso di un Enterprise Service Bus (ESB).

1.2.1 Gli Enterprise Service Bus

La realizzazione della SOA basata sui WS porta alla creazione di molte comunicazioni punto-a-punto, rendendo spesso l'intera infrastruttura difficile da manutenere a fronte di cambiamenti nei servizi stessi. Infatti, in questo modello, se cambia anche solo il protocollo per accedere ad un servizio è necessario modificare tutti i componenti che dipendono da quel servizio. Per questo motivo, più un'organizzazione abbraccia il paradigma SOA più sentirà la necessità di un'infrastruttura che, da un lato, renda uniforme l'accesso ai servizi, e, dall'altro, possa essere impiegata per utilizzi più sofisticati dei servizi stessi.

L'infrastruttura sopra delineata viene chiamata Enterprise Service Bus (ESB) ed, infatti, è un vero e proprio bus di comunicazione fra i vari servizi, più correttamente è un middleware capace di integrare fra loro i servizi in una SOA. Infatti gli ESB vengono utilizzati per connettere le più disparate applicazioni e risorse, mediando le loro incompatibilità, orchestrando la loro interazione e rendendole utilizzabili per scopi futuri. L'ESB svolge quindi un ruolo di primaria importanza all'interno della SOA, poiché agisce sia come

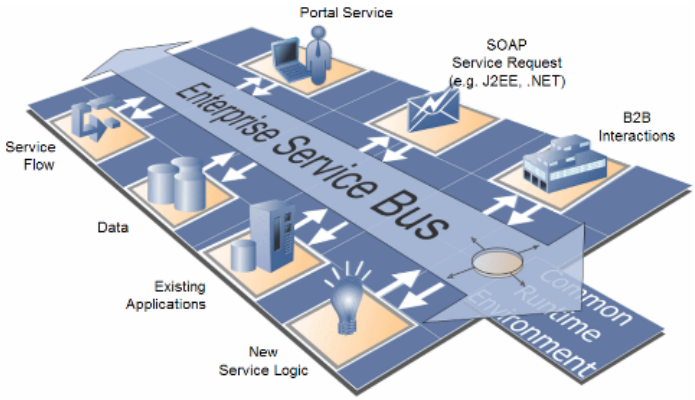


Figura 1.2: Visione d'alto livello di un Enterprise Service Bus

un registro di servizi sia come un unico punto centralizzato di gestione.

Gli Enterprise Service Bus hanno inoltre il grande compito di uniformare l'accesso ai servizi, in particolare soluzioni middleware preesistenti e sistemi legacy: gli ESB, infatti, rendono accessibili tutti gli applicativi in modo assolutamente omogeneo e coerente con il modello basato sui WS. Mediante l'introduzione di un ESB, tutte le comunicazioni fra i servizi vengono effettuate attraverso di esso in modo assolutamente trasparente agli stessi servizi: è addirittura possibile trasformare i messaggi prima che questi vengano effettivamente consegnati, consentendo una normalizzazione utile durante l"intera esecuzione dei processi di business. Questa caratteristica è di primaria importanza, poiché in questo modo è possibile evitare la propagazione dal produttore al consumatore di eventuali modifiche: ad esempio, se cambiasse il formato dei messaggi in ingresso ad un servizio, basterebbe introdurre, all'interno dell'ESB, una trasformazione dal vecchio al nuovo formato.

1.3 Descrizione del progetto Pitagora

Basandosi in larga parte sul paradigma e l'infrastruttura precedentemente descritta, è stato progettato e realizzato il Progetto PITAGORA, una piattaforma di gestione e supervisione aeroportuale che qui è brevemente presentata. Il lavoro di tesi è stato effettuato nell'ambito di tale progetto ed ha interessato una piccola porzione di esso, relativa ad aspetti di sicurezza.

1.3.1 Scenario di progetto

Gli aeroporti sono scenari complessi, con una grande varietà di persone appartenenti a differenti unità di business o società: oggigiorno, la gestione aeroportuale ha raggiunto una maturità adeguata per affrontare tale complessità, nei casi in cui operazioni e procedure siano state pianificate in anticipo e gli aeroporti siano in funzione in condizioni regolari o condizioni irregolari previste. Tuttavia, al verificarsi di situazioni anormali o non pianificate, gli aeroporti riscontrano difficoltà ad essere gestiti efficientemente e le conseguenze possono essere non critiche, come un blocco aeroportuale con un negativo impatto economico, fino a situazioni molto critiche come il rischio di perdita di vite .

Per affrontare queste situazioni e tutte le restanti problematiche legate ad un'infrastruttura critica come un aeroporto, nasce il Progetto PITAGORA (Piattaforma Integrata per la Gestione delle Operazioni Aeroportuali). Il progetto, finanziato dalla Regione Toscana, vede la partecipazione di 8 partner (1 Group Leader, 5 PMI e 2 Università) e rivolge l'attenzione a quelli che sono comunemente identificati come i principali problemi nella gestione di un

aeroporto: collaborazione, risorse e crisi. Mediante più fasi di progettazione, sviluppo, prototipazione e testing, il progetto mira a creare una piattaforma per la gestione integrata e efficiente di un'infrastruttura aeroportuale.

1.3.2 Obiettivi

In particolare, lo scopo del progetto è costruire una piattaforma per l'integrazione di sistemi eterogenei in modo da migliorare l'operatività dell'aeroporto, in termini di:

- Operatività di voli;
- Gestione bagagli
- Gestione ed incremento della sicurezza;
- Prevenzione e risoluzione di incidenti;
- Amministrazione delle risorse aeroportuali.

Grazie a tale piattaforma, l'aeroporto sarà in grado di gestire tutti i sistemi, interni alla struttura, in modo integrato; infatti, il progetto ha l'obiettivo di creare un punto di dialogo tra l'aeroporto e gli utenti, ad esempio tramite l'uso di applicazioni smartphone, in modo da acquisire informazioni sul loro grado di soddisfazione, gestire l'insoddisfazione ed accrescere il grado di apprezzamento dei passeggeri.

1.3.3 Risultati attesi

Il supporto che la piattaforma PITAGORA offre in termini di miglioramento, garantirà:

- un incremento della capacità operativa dell'aeroporto in termini di flusso nel traffico aereo;
- una più efficiente gestione energetica e del personale operativo, ponderata in tempo reale su scenari e carichi operativi;
- un'individuazione degli indicatori più critici ai fini della gestione dei processi ed una conseguente ottimizzazione della gestione stessa in base ai parametri identificati;
- un tempo di reazione minimo in caso di crisi con precise indicazioni su come gestire lo scenario che di volta in volta si presenta.

1.4 Design dell'architettura

1.4.1 Business Architectural Design

Le principali funzioni di un Aeroporto sono la gestione dei **passeggeri**, degli **aeromobili** e dei **beni** che vi transitano, garantendo alti standard di sicurezza e ottimizzando costi e ricavi. All'aumentare della complessità, assume sempre più importanza la possibilità di disporre del supporto di una piattaforma per l'integrazione, la comunicazione e l'automatizzazione dei dati, che utilizzi e valorizzi l'abilità dei dipendenti nella gestione delle attività ordinarie e non. Le funzionalità e le operazioni di business che la piattaforma dovrà supportare in merito alla gestione dell'aeroporto sono state identificate dal Team di progettazione e sviluppo in:

- **Collaboration** - per mezzo della collaborazione all'interno della community aeroportuale, superando i confini relativi alle diverse unità di business, le procedure risulteranno più semplici e i costi ridotti. Obiettivo finale di tale collaborazione è garantire un servizio ottimale al passeggero, minimizzando le inefficienze. La piattaforma implementerà A-CDM così come definita da Eurocontrol.[1]

- **IT Aided Safety and Security** - comprende la gestione di incidenti e crisi, a partire dal monitoraggio di allarmi a bassa priorità fino alla gestione delle operazioni da effettuare in caso di crisi. Facilita così il lavoro dei supervisori e contribuisce a scovare cosiddette "zone ombra" in cui potrebbero celarsi minacce non previste.

- **Reporting and Analysis** - si fa uso di analisi di particolari indicatori al fine di migliorare le prestazioni e l'efficienza operativa aeroportuale.

- **Health and Status monitoring** - fornisce una conoscenza globale relativamente alle strutture e ai sistemi tecnici in uso. Le informazioni sullo stato di salute tecnico coprono l'intero aeroporto e in base ad esse è possibile garantire e prevedere l'integrità funzionale.

- **Automatic Business Processing enabling** - permette di alleggerire il carico di lavoro degli operatori, eseguendo autonomamente i possibili processi e per i restanti, per i quali è prevista l'attenzione dell'operatore, li assiste supportandone le decisioni.

[1] http://www.eurocontrol.int/services/airport-collaborative-decision-making-cdm-0

- **Communication technologies potentiality** - permette di sfruttare il potenziale dato dalla enorme diffusione delle tecnologie IT, quali smartphone o tablet, tramite l'uso di servizi dedicati al passeggero che facciano uso di questi sistemi.

- **Passenger Experience Enhancement** - estende l'uso dei servizi al passeggero fornendo nuove funzioni attrattive o finalizzate al comfort.

1.4.2 System Architectural Design

La piattaforma, come già accennato, è progettata secondo i criteri della Service Oriented Architecture: il design del sistema è realizzato secondo il concetto dei Servizi e delle loro interfacce. Il concetto di servizio è qui visto come punto di congiunzione tra il business e lo sviluppo, due aspetti distinti ma qui complementari:

- Il business considera il servizio in funzione delle persone, dei processi e della tecnologia.

- Lo sviluppo considera il servizio in termini di codice applicativo ed interfacce.

E' stata quindi seguita la "Open Group's SOA Reference Architecture" come modello per la classificazione e la stratificazione dei servizi. La Reference Architecture, mostrata in Figura 1.3, è un fattore chiave per il raggiungimento del valore atteso da parte di una SOA.

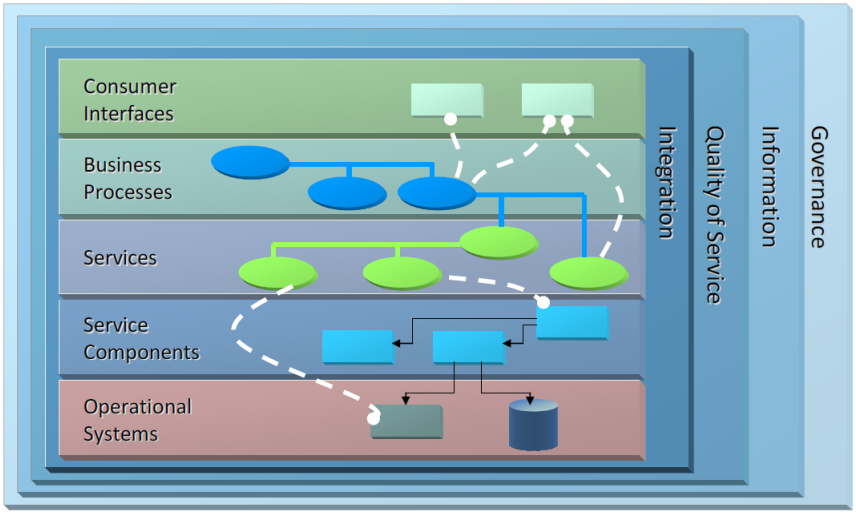


Figura 1.3: SOA Reference Architecture

La Figura 1.4 mostra, invece, una visione ad alto livello dell'intera piattaforma, dando un'idea delle interconnessioni tra i differenti moduli, il middleware e la console dell'operatore.

Le funzioni offerte sono rese disponibili ai client connessi al server via TCP/IP network. Grazie all'uso di tecnologie web, i client non avranno bisogno di installare sulle loro macchine alcun software specifico ma solo un Web browser e tutti gli aggiornamenti relativi al sistema operativo in uso. Per ognuno dei singoli moduli è possibile fornire una breve descrizione delle funzioni che assolverà:

- COLLABORATION LOGICAL MODULE: è pensato per permettere e migliorare la collaborazione tra le parti (gli stakeholders) che utilizzano la piattaforma PITAGORA. Collezionando dati da differenti sistemi

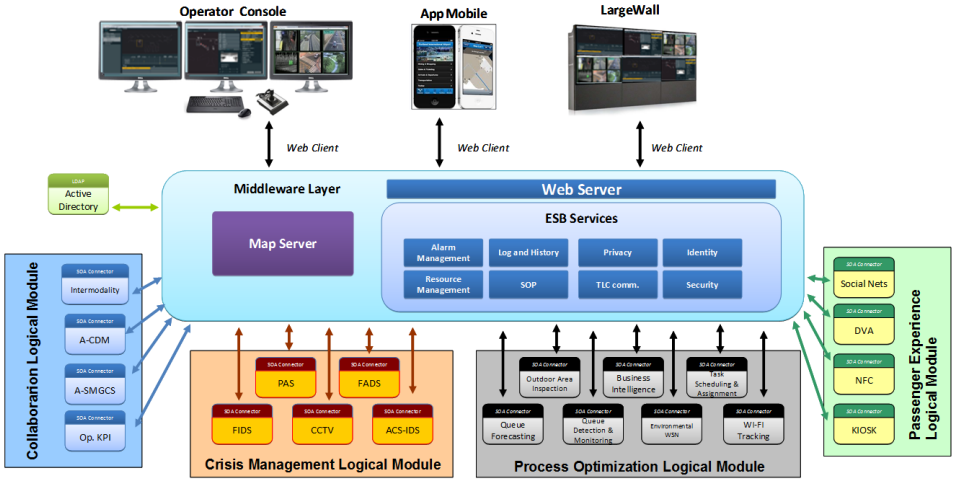


Figura 1.4: Architettura ad alto livello dell'intera Piattaforma PITAGORA

aeroportuale si propone di garantire: un aumento della consapevolezza da parte di tutti gli interessati e un ottimizzazione nello svolgimento delle procedure.

- CRISIS MANAGEMENT LOGICAL MODULE: opera in background e si attiva in caso di situazioni critiche per il business o per la sicurezza dei passeggeri. Consiste nel monitoraggio dell'ambiente aeroportuale, nell'individuazione di eventi critici, nella chiamata delle squadre di soccorso e nella gestione della crisi.

- PROCESS OPTIMIZATION LOGICAL MODULE: consiste in una serie di sotto moduli pensati per il monitoraggio e l'ottimizzazione dei processi legati alle attività aeroportuali.

- PASSENGER EXPERIECE LOGICAL MODULE: modulo molto importante, ha come obiettivo il miglioramento della soddisfazione dei

passeggeri e sfrutta ad esempio reti sociali o sistemi mobile per interfacciarsi con essi.

Capitolo 2

Panoramica delle tecnologie utilizzate

Prima di discutere degli aspetti di sicurezza saranno qui esposte in breve una serie di tecnologie utilizzate nell'ambito dello svolgimento del lavoro di tesi. Tali tecnologie sono alla base del progetto PITAGORA e con esse è stato realizzato un modello (un prototipo) semplificato come base su cui operare. In particolar modo si inizierà facendo riferimento ad alcuni concetti di *modularità* legati alla specifica OSGi per poi passare alla descrizione dell'ESB utilizzato e alle tecnologie ad esso legate (ad esempio CAMEL). Infine si parlerà di Web Services, di fatto la più diffusa concretizzazione di servizio SOA, poiché ne è stato implementato uno internamente all'ESB per realizzare la comunicazione tra i sottomoduli dei partner e la piattaforma.

2.1 OSGi: Open Service Gatway initiave

Per capire cosa sia e che uso venga fatto della specifica OSGi è prima necessario introdurre un concetto, noto e spesso abusato nel mondo dell'informatica, ossia quello della modularità del software.

2.1.1 Un'introduzione alla modularità

Iniziamo col dire che esiste una **modularità** a **run-time** ed una a **build-time**. Ci concentreremo sulla definizione di modularità a run-time poiche' l'OSGi offre questo tipo di funzionalità.

> "A software module is a deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers"

I moduli, a differenza di altre entità come le classi o i package, sono unità discrete della distribuzione che possono essere affiancate da altri moduli software. Cioè i moduli sono entità più fisiche rispetto alle classi o ai package. Un esempio di modulo può essere un file **JAR**.[3]

Un modulo può essere gestito nel senso che può essere installato, disinstallato o aggiornato. Durante le fasi di sviluppo di un applicazione software, la divisione in moduli può aiutare a pianificare e a parallelizzare le attività.

Un modulo è un'**entità testabile indipendente** ed è pensato per essere riusato da più processi, ma le operazioni che espone possono essere invocate solo da chiamate dirette a metodo.

I moduli sono unità **componibili** tra loro. Può succedere che un macro-modulo sia composto da più micro-moduli.

Rendere un qualsiasi processo o prodotto una composizione di moduli ha degli importanti vantaggi tra i quali la diversificazione dei ruoli tra i moduli stessi, l'astrazione del modulo di cui si identifica solo il compito che espleta e non l'implementazione, il riutilizzo del modulo in diversi contesti nonché, cosa non di poco conto, la facilità di manutenzione.

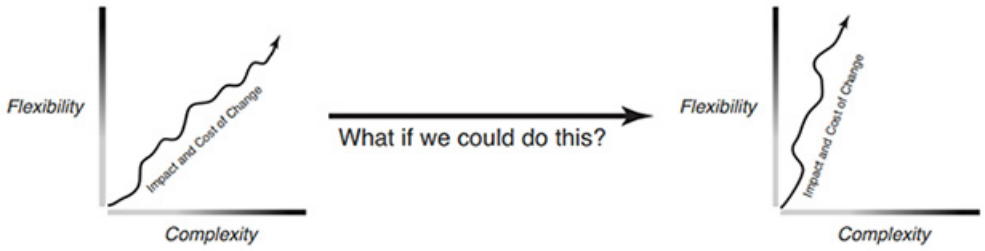


Figura 2.1: Flessibilità contro Complessità

Il grafico in Figura 2.1 riflette il compromesso tra flessibilità e complessità che, la modularità aiuta a superare. Per avere un'idea più chiara è possibile basarsi su un esempio pratico: supponiamo di dover scrivere un software che permetta di applicare modifiche future in maniera semplice e con pochi costi aggiuntivi. Quello che voglio è che questo software sia il più flessibile possibile, ma la **flessibilità** ha un costo che si paga con la **complessità**. Siamo in presenza di un paradosso: per ridurre i costi delle modifiche al mio software dovrei renderlo più flessibile, ma così facendo lo rendo più complesso e quindi aumento i costi di manutenzione. Il grafico mostra come, introducendo il concetto di modularità nel software, la pendenza della curva cambia riducendo la complessità a parità di flessibilità. Questo ragionamento si conclude con l'assioma: *"Modularity is the best hope to reduce the cost of changes due*

to architectural designing".

2.1.2 Il framework OSGi

Lo standard OSGi nasce dal lavoro svolto dall'OSGi Alliance, un'organizzazione fondata nel 1999 da Ericsson, IBM, Oracle e altri partner minori, che "propone un processo collaudato e maturo per creare specifiche aperte che permettono una composizione modulare del software integrato con la tecnologia Java".[4] Il framework mira ad implementare un modello a componenti, completo e dinamico, risolvendo molti dei problemi legati allo scarso supporto di Java nella modularità e nel dinamismo attraverso alcuni concetti fondamentali:

- Definizione del concetto di modulo (chiamato *bundle*)
- Gestione automatica delle dipendenze
- Gestione del ciclo di vita del codice (*Lifecycle management*)

I problemi che OSGi si propone di risolvere sono dovuti principalmente al sistema di class loading di Java e della modalità di distribuzione e deploy dei pacchetti, ossia i JAR.

In un programma Java, i JAR sono considerati come una lista di classi globale, il famoso classpath. Le classi necessarie quindi sono considerate solo a deploy-time e build-time; non esiste un concetto di JAR a run-time. La gestione delle dipendenze tra JAR è inoltre un altro problema non di poco conto, essendo assente uno standard per poterle specificare.

Manca, infine, in Java un meccanismo per permettere il mascheramento di informazioni tra archivi JAR. I modificatori di accesso come public, protected, private, default si riferiscono alla visibilità tra packages e non tra JAR. Spesso capita che classi facenti parte di JAR abbiano bisogno di avere accesso a classi di altri package dello stesso JAR. Questo significa che lo sviluppatore è costretto a usare il modificatore public perdendo il controllo sul mascheramento dei dati.

E' ovvio quindi che i JAR, così come intesi dalla specifica Java, non possono essere considerati dei veri e propri moduli in quanto sono fortemente accoppiati e difficilmente indipendenti tra di loro.

La tecnologia OSGi viene in aiuto degli sviluppatori, rendendo un archivio JAR un componente indipendente che possiede tutte le caratteristiche di modularità; tale componente prende il nome di **bundle**.

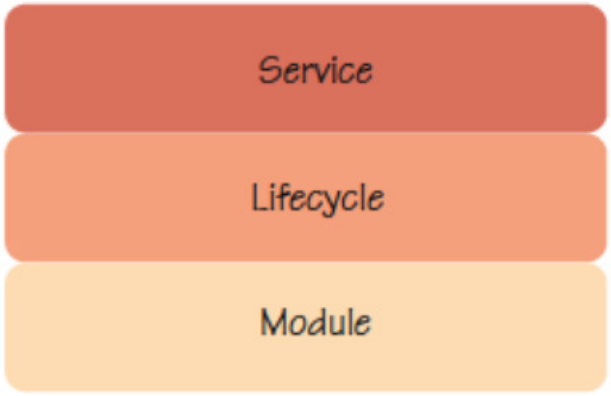


Figura 2.2: Livelli dell'architettura OSGi

A questo punto non resta che spiegare la struttura del framework, visibile in Figura 2.2, così da capire come ognuno dei suoi layer vada a colmare le problematiche precedentemente esposte.

2.1.2.1 Module Layer

Il primo strato è il *Module layer*. Le specifiche definiscono l'entità atomica della modularità: il **bundle**. Il bundle è un file JAR con delle caratteristiche speciali.

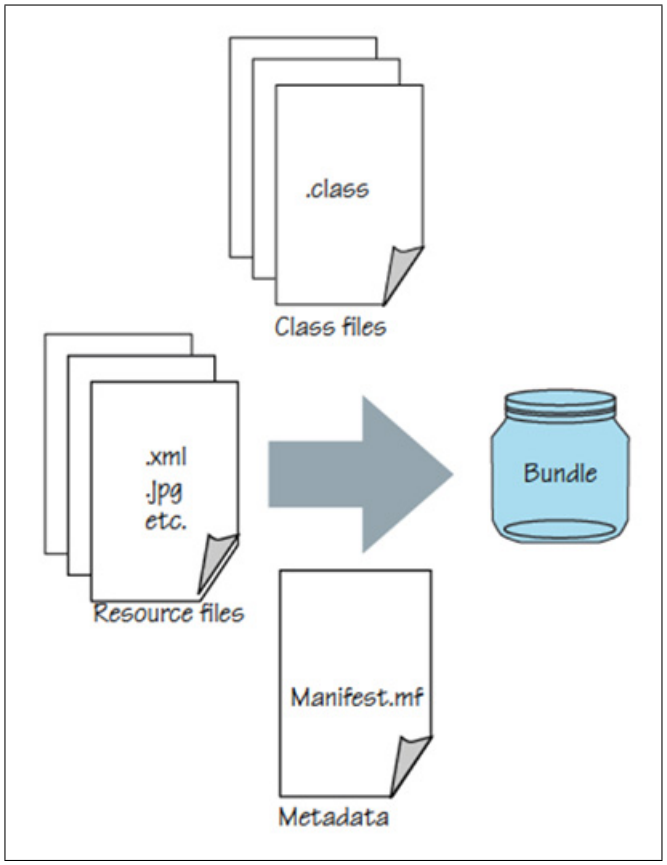


Figura 2.3: Componenti del bundle

Dalla Figura 2.3 osserviamo che un bundle può contiene: *sorgenti*, *risorse* e il *manifest*. La prima differenza sostanziale tra un bundle e un JAR e che il primo ha un **classpath limitato a se stesso**, cioè un bundle vede solo le classi definite al suo interno. Sebbene questa possa apparire come una limitazione in realtà è un vantaggio e possiamo dedurlo analizzando la seconda differenza.

Il file **manifest** (Figura 2.4) è arricchito, rispetto a quello di un JAR standard, con headers aggiuntivi raggruppati secondo le specifiche OSGi in categorie quali **identificazione** e **descrizione**, **classloading** e **attivazione**.

```
Manifest-Version: 1.0 Bnd-LastModified: 1386750447262 Build-Jdk: 1.7.0_40 Built-By:
gdg-firenze Bundle-ManifestVersion: 2 Bundle-Name: GDG Firenze :: Sensormix ::
Example Bundle Bundle-SymbolicName: example-bundle Bundle-Vendor: GDG Firenze ::
Sensormix Team Bundle-Version: 1.0.0.SNAPSHOT Bundle-Activator:
com.google.developers.gdgfirenze.dataservice.Activator
Created-By: Apache Maven Bundle Plugin Export-Package:
com.google.developers.gdgfirenze.model;version="1.0.0.SNAPSHOT",
com.google.developers.gdgfirenze.osgi;version="1.0.0.SNAPSHOT",
com.google.developers.gdgfirenze.service;version="1.0.0.SNAPSHOT" Import-Package:
javax.jws,javax.jws.soap,javax.xml.bind.annotation,javax.xml.ws Tool: Bnd-1.50.0
```

Figura 2.4: Esempio di file Manifest di un bundle

La categoria identificazione e descrizione comprende header quali Bundle-Name e Bundle-Version che si usano per definire il nome del bundle e la sua versione. La categoria attivazione comprende header come il Bundle-Activator che permette di definire quale classe all'interno del bundle sarà avviata automaticamente durante la fase di start-up del bundle. Infine la categoria di classloading comprende header quali Export-Package e Import-

Package e permette di definire quali package mettere a disposizione dei bundle esterni e soprattutto quali package devono essere importati perché necessari alla vita del bundle. In sostanza, tramite il manifest, possiamo estendere la visibilità del classloading del bundle, permettendo l'utilizzo di classi definite in altri moduli. Ecco la svolta: per poter usare classi di altri bundle devo definire esplicitamente la dipendenza. Questo concetto, per quanto semplice, è estremamente importante perché permette di definire con precisione i confini del bundle.

2.1.2.2 Lifecycle Layer

Il secondo strato è caratterizzato da una serie di API che definiscono il **ciclo di vita** del bundle, secondo il diagramma di flusso mostrato in Figura 2.5.

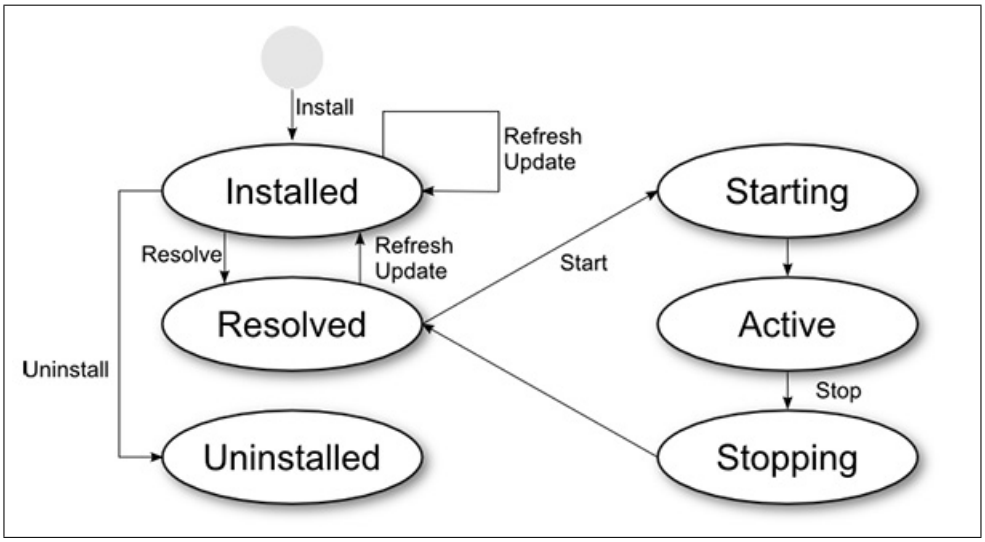


Figura 2.5: Il ciclo di vita di un bundle

Dopo aver installato il bundle, il framework analizza il **manifest** alla ricerca degli headers della categoria **identificazione e descrizione**. Qui vengono fatti check di congruenza e di versione. Se l'identificazione è corretta, il bundle va in stato **Installed**. Il framework, a questo punto, ispeziona il manifest alla ricerca della sezione di classloading. In questa fase vengono risolte le dipendenze dichiarate nel manifest verificando che i package definiti nell'header **Import-Package** siano disponibili. Solo quando tutte le dipendenze del bundle sono state risolte, il bundle avanza nello stato successivo, ossia in stato **Resolved**. A questo punto il framework analizza la categoria **attivazione** per allocare un'istanza della classe che implementa l'interfaccia **BundleActivator**, passa lo stato del bundle in **Starting** e chiama il metodo **start()**. Se tale metodo termina senza eccezioni lo stato del bundle diviene **Active**. L'interfaccia **BundleActivator** definisce anche il metodo **stop()** che sarà invocato quando richiesto. Se lo **stop()** va a buon fine il bundle ritorna in stato **Resolved**.

Le API del Lifecycle layer definiscono anche il concetto di **BundleContext** che permette al bundle di interagire con il framework non solo per ciò che riguarda il ciclo di vita del bundle ma soprattutto per la gestione dei servizi

2.1.2.3 Service Layer

L'ultimo strato, il Service Layer, è caratterizzato da API che non si occupano di definire un Servizio (la filosofia legata al servizio è la stessa definita per le SOA) ma piuttosto definiscono il ciclo di vita dei servizi e il modo in cui interagiscono tra di loro.

```
public interface FileManagerService {
                    void open(String filename);
                    void save();
                    void close();
          }
```

Supponiamo di aver definito un'interfaccia Java che espone dei metodi **open**, **save**, **print** e di aver realizzato l'implementazione di questa interfaccia che applica queste operazioni ad un file in formato ASCII (.txt). Ora vogliamo che altri oggetti, magari scritti da altri o che ancora non sono stati scritti, possano usufruire delle funzionalità offerte dal nostro servizio. A questo punto gli altri bundle, per vedere la nostra interfaccia, dovranno aver dichiarato espressamente la dipendenza nel loro manifest. Buona norma vuole che non si creino dipendenze tra un bundle che espone un servizio e un altro che lo usa, quindi quello che si fa è definire l'interfaccia in un bundle separato. In questo modo gli altri bundle potranno utilizzare i metodi definiti nell'interfaccia senza avere dipendenze dirette dal bundle che contiene l'implementazione.

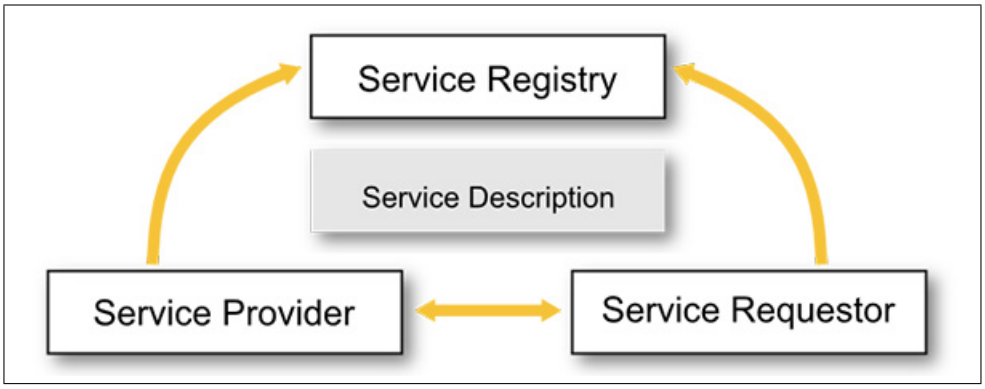


Figura 2.6: Modello service oriented

A questo punto si pone il problema di *come ottenere a runtime un'istanza del servizio*. Il Service Layer offre degli strumenti per risolvere questo problema.

Il **Service Provider**, ossia bundle che implementa l'interfaccia Java da noi definita, registrerà l'istanza del servizio al **Service Registry** utilizzando il metodo *registerService* del BundleContext, che come abbiamo detto prima, fornisce delle funzionalità per interagire con il framework. Il **Service Requestor**, ossia un altro bundle che vorrà usare la suddetta implementazione, richiederà, quando necessario, l'istanza del servizio al Service Registry. Tutte le interazioni ruotano attorno al concetto di **Service Description**, ossia l'interfaccia del servizio.

Le richieste possono essere di vario tipo a seconda delle esigenze. Per esempio, nel caso in cui il Service Registry non conoscesse l'istanza per una specifica Description, sarà possibile gestire in diversi modi lo scenario. Se la funzionalità richiesta è necessaria per un'operazione real time, sarà possibile programmare il bundle in modo da lanciare un'eccezione. L'eccezione potrà essere gestita così da non compromettere l'esecuzione dell'applicazione. Se invece la funzionalità è richiesta per un'operazione batch, si potrà decidere di aspettare fintanto che il servizio non sarà disponibile.

2.2 ServiceMix

Apache ServiceMix è un ESB che combina le funzionalità di una SOA ai concetti di modularità.[5] L'adozione di un bus permette di disaccoppiare le applicazioni e dunque ridurne le dipendenze. I dati transitano sul bus e sono

instradati verso i servizi sotto forma di "messaggi". Il bus inoltre supporta differenti protocolli e modalità di comunicazione per lo scambio di tali dati con le applicazioni esterne.

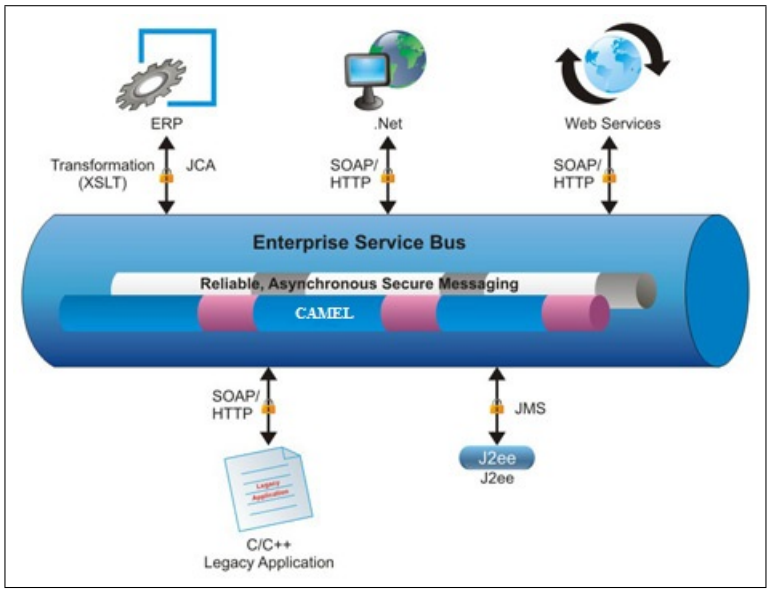


Figura 2.7: Enterprise Service Bus

La panoramica fin qui fatta in merito ad OSGi ci è utile a capire come tale framework si lega all'ESB in questione. Gli sviluppi di ServiceMix iniziano intorno al 2005 col fine di realizzare una implementazione open source delle specifiche **Java Business Integration (JBI)**, la JSR 208. Per cercare però di risolvere proprio problemi dovuti a requisiti della JBI, nel 2008 il team decide di provare la strada **OSGi** e quindi cominciano gli sviluppi del **ServiceMix Kernel**, ovvero un container basato su OSGi che sarà poi la base di ServiceMix dalla successiva major release, la 4.0. Ben presto questo nuovo container cresce a tal punto da meritare attenzione più generale,

assume il nome di **Karaf** e si sposta fino a diventare un progetto Apache di primo livello.

ServiceMix infatti è logicamente composto da una serie di componenti (vedi Figura 2.8), ma tra questi concentriamo il nostro interesse su due in particolare: **Apache Karaf** e **Apache Camel**.

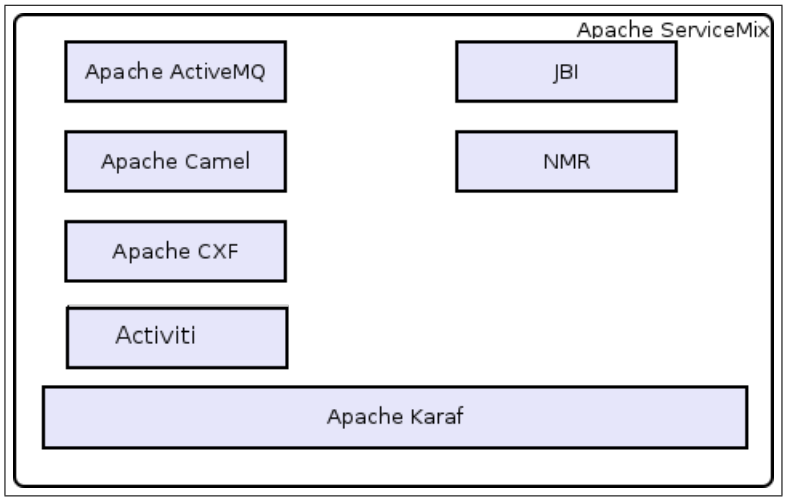


Figura 2.8: Rappresentazione a componenti di ServiceMix

2.2.1 Apache Karaf: alias ServiceMix Kernel

Karaf è un "ambiente runtime basato su OSGi che fornisce un container leggero nel quale poter installare componenti e applicazioni" in Java. Ciò significa che si possono scrivere delle applicazioni o delle librerie, impacchettarle dentro ad un JAR (che in OSGi sono chiamati ufficialmente Bundle) e installarle dentro a Karaf come se fosse un sistema operativo. Karaf NON è però un'implementazione della specifica OSGi bensì, supporta una delle pos-

sibili implementazioni (di default Apache Felix) ed in più fornisce a contorno una serie di strumenti utili per amministrare un sistema di questo genere.

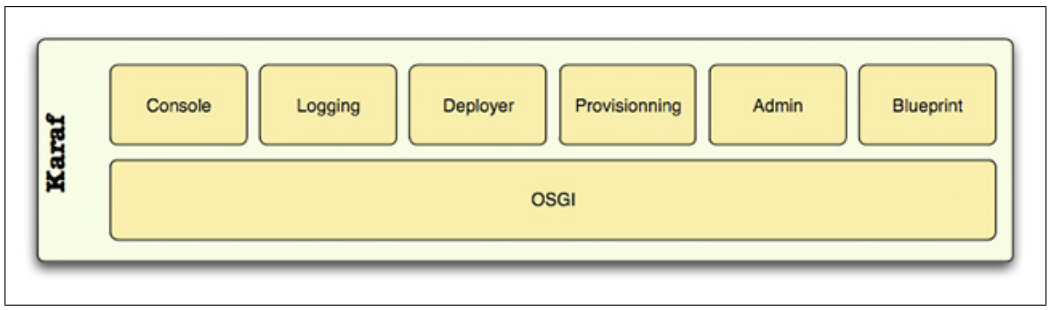


Figura 2.9: Architettura di Karaf

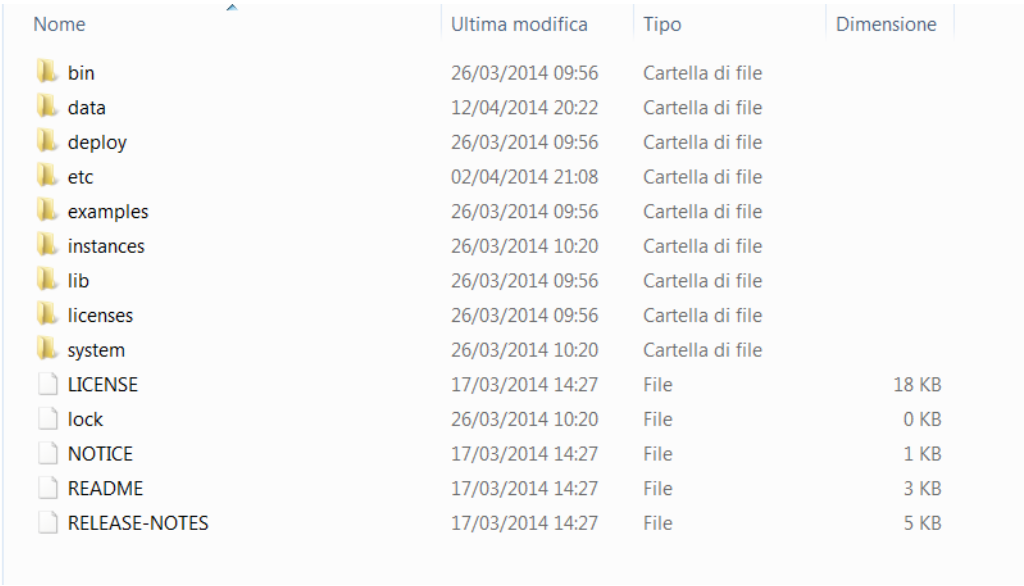
Tra gli strumenti più importanti forniti da Karaf citiamo la possibilità di **installare a caldo** nuovi bundle, la **configurazione dinamica** (che offre un registro di sistema per i moduli), l'accesso remoto tramite **SSH** direttamente alla console OSGi ed una **console web** attraverso la quale amministrare il sistema.

Nel seguente paragrafo si faranno alcuni accenni di carattere più pratico, utili per orientarsi in seguito nell'ambito della soluzione proposta nel seguente lavoro.

2.2.2 ServiceMix in pratica

2.2.2.1 La struttura di ServiceMix

Una volta scaricata l'ultima versione di ServiceMix, senza alcun bisogno di installazione, la si può semplicemente decomprimere ottenendo un'alberatura dei file estratti come in Figura 2.10



Nome	Ultima modifica	Tipo	Dimensione
bin	26/03/2014 09:56	Cartella di file	
data	12/04/2014 20:22	Cartella di file	
deploy	26/03/2014 09:56	Cartella di file	
etc	02/04/2014 21:08	Cartella di file	
examples	26/03/2014 09:56	Cartella di file	
instances	26/03/2014 10:20	Cartella di file	
lib	26/03/2014 09:56	Cartella di file	
licenses	26/03/2014 09:56	Cartella di file	
system	26/03/2014 10:20	Cartella di file	
LICENSE	17/03/2014 14:27	File	18 KB
lock	26/03/2014 10:20	File	0 KB
NOTICE	17/03/2014 14:27	File	1 KB
README	17/03/2014 14:27	File	3 KB
RELEASE-NOTES	17/03/2014 14:27	File	5 KB

Figura 2.10: Elenco directory ServiceMix

Tra le directory principali elenchiamo:

- **bin**: contiene i comandi, tra cui quello per avviare ServiceMix
- **etc**: contiene i file di configurazione
- **system**: contiene alcuni bundle di sistema di Karaf (framework, logging, etc.)
- **deploy**: è la directory più intrigante, permette infatti "l'installazione a caldo" semplicemente copiandovi dentro un JAR.

2.2.2.2 Karaf Console

Avviando ServiceMix, quello che sostanzialmente stiamo avviando è il container Karaf. Una volta avviato Karaf, l'implementazione del servizio Confi-

guration Admin tiene sotto controllo i file che si trovano nella cartella *etc* e, utilizzando le API del Lifecycle Layer, può notificare ai bundle le modifiche alla configurazione. Da Windows, portandosi nella directory *bin* e usando il comando "servicemix" si avvierà il container come in Figura 2.11

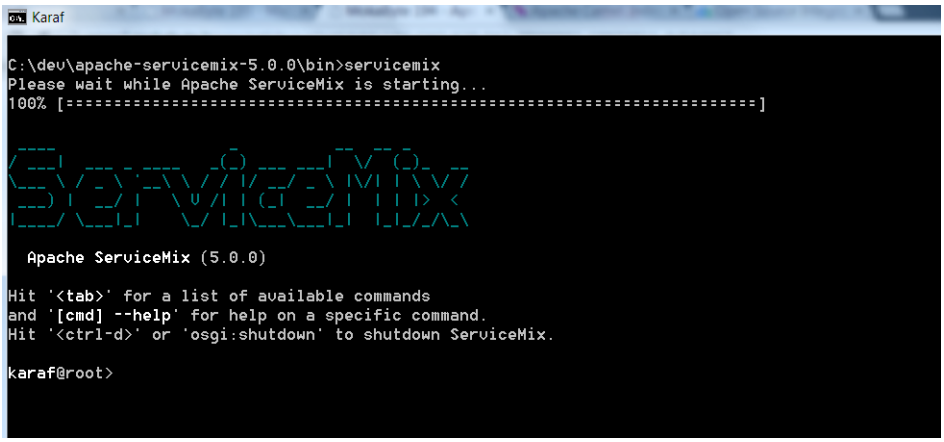


Figura 2.11: Console di ServiceMix

Abbiamo la possibilità di interagire con la piattaforma per mezzo della console, sfruttando una serie di comandi; tra i più rilevanti:

- **osgi**: raggruppa l'inseme dei comandi di più basso livello per l'interazione con OSGi;
- **config**: è l'insieme dei comandi per interagire con il servizio di configurazione dichiarando nuove variabili di configurazione o modificando il valore di variabili esistenti;
- **admin**: raggruppa i comandi per la gestione della console;

- **features**: è l'insieme dei comandi che permettono di gestire le features, una funzionalità molto utile di Karaf.

Ad esempio tramite il comando **osgi:list -t 0** si può mostrare la lista di tutti i bundle installati, compresi quelli di sistema (vedi Figura 2.12).

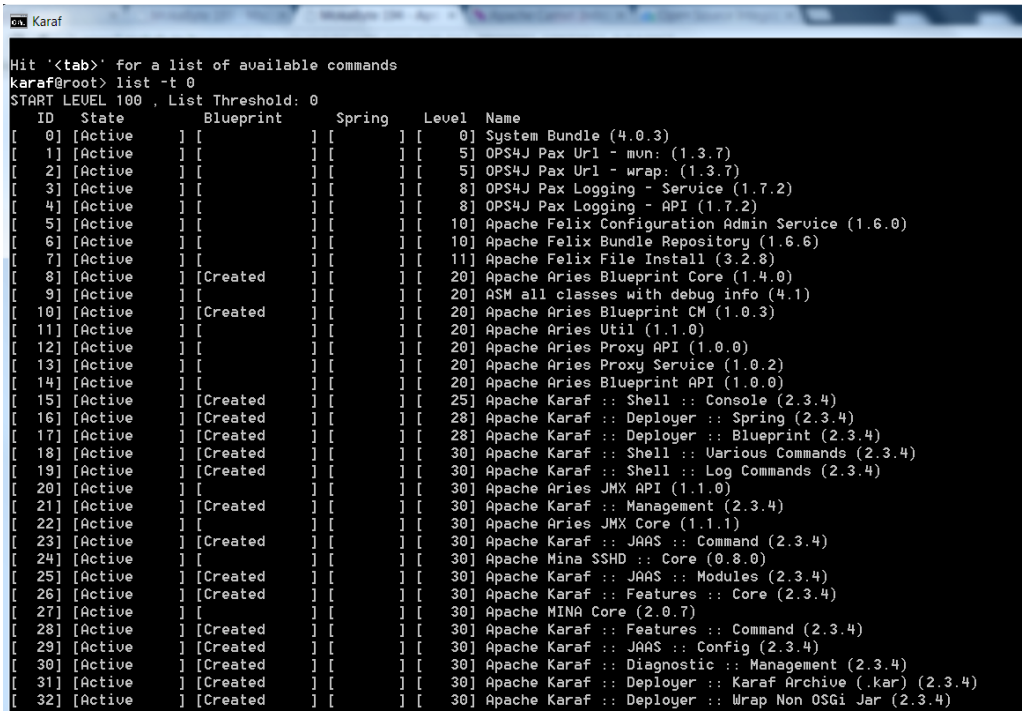


Figura 2.12: Lista completa dei bundle installati

Analizzando sempre la Figura 2.12) si osservi il significato delle varie colonne. La colonna **id** rappresenta l'identificato del bundle, la colonna **State** rispecchia il ciclo di vita del bundle precedentemente esposto e riconducibile alla Figura 2.5. Le colonne **Blueprint** e **Spring** mostrano lo stato dei servizi creati utilizzando una delle due estensioni al framework. La colonna **Level** dà

indicazione circa l'ordine con cui il bundle viene elaborato, ossia processato all'avvio del container, ed infine **Name** è il nome del bundle indicato nel Manifest alla voce Bundle-Name.

2.2.2.3 Features e Hot deploy

Senza dilungarci troppo a riguardo, è possibile inoltre utilizzare il comando **features**, utile per installare uno o più bundle insieme. L'uso delle features si lega molto bene con strumenti di "automazione dello sviluppo" come Maven; dopo aver configurato correttamente il local repository Maven in uso tra le configurazioni di Karaf, si potranno installare in modo molto semplice i JAR che stiamo sviluppando. Una delle feature più immediate è la **Web Console** che, una volta installata tramite comando **features:install webconsole** è subito raggiungibile da browser all'indirizzo **http://localhost:8181/system/console/**. Fornisce una GUI per un'interazione più user-friendly.

Ma non è tutto: se, per esempio, siamo in fase di deploy e abbiamo bisogno di installare un bundle per provarlo e magari di sovrascriverlo più volte per risolvere i bug che riscontriamo, Karaf ci permette di copiare i bundle (i JAR) nella cartella deploy e di installarli al volo. E' questa la funzionalità di hot deploy.

2.2.3 Una serie di tecnologie di contorno

2.2.3.1 Spring e Blueprint

Spring è il famoso framework open source che fa quasi tutto, ma in questo caso ci interessa la sua base, ossia la Dependency Injection e in particolare la possibilità di istanziare le classi della nostra applicazione Java a partire da file di configurazione XML. Karaf esplora i JAR che vengono installati alla ricerca di questi file e ne crea i bean che vi sono definiti dentro quando li trova. In particolare, il file XML per la configurazione Spring dovrà essere localizzato rispetto al classpath del progetto, nella directory resources/META-INF/spring/nomefile.xml. Sarà possibile capire nel seguito come questo meccanismo, ossia l'uso di file di configurazione sia molto vantaggioso.

2.2.3.2 PAX Web

Karaf include al suo interno un Web Server e che su si esso si possono installare delle applicazioni web. Il framework Pax Web fornisce a Karaf il supporto per fare web, e in particolar modo rivedremo come questo WebServer verrà configurato in fase di sviluppo della soluzione per una corretta comunicazione tra la piattaforma e le applicazioni dei partner.

2.2.3.3 Apache CAMEL

Apache Camel è il secondo importante tassello che entra in gioco insieme a Karaf. CAMEL è un toolkit per sviluppare **integrazione** ovvero "scambiare informazioni tra applicazioni"[6] ed entra in gioco laddove queste utilizzano

protocolli, formati dati e interfacce non compatibili tra di loro. Il framework si basa sul paradigma dei **messaggi**, scambiati tra due **endpoint** attraverso delle **rotte** lungo le quali il messaggio è processato, trasformato, adattato e instradato [7].

CAMEL fornisce:

- un'implementazione dei più noti **EIP (Enterprise Integration Patterns)**;
- connettività in termini di supporto di una gran varietà di protocolli;
- definizione di uno speciale *Domain Specific Language* (DSL), disponibile sia in forma Java che in forma XML.

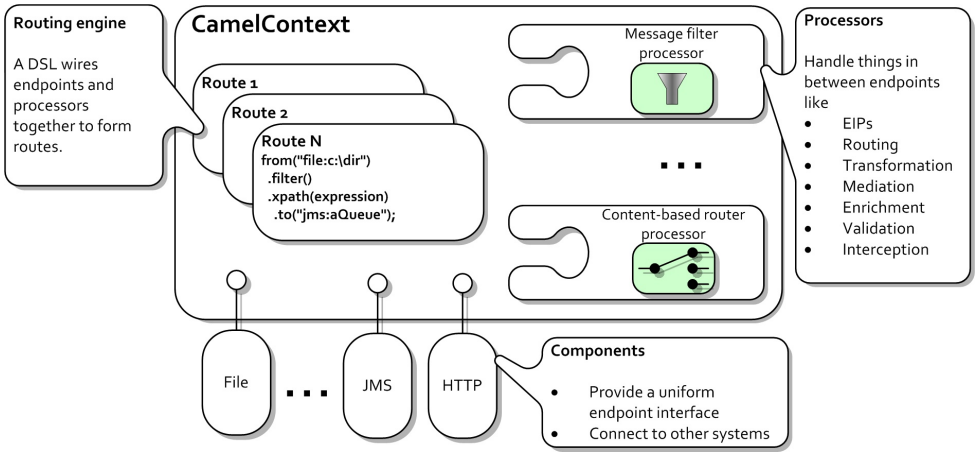


Figura 2.13: Architettura CAMEL

La Figura 2.13 mostra come questi tre elementi si vadano a mappare su una serie di strumenti propri di CAMEL: i **Component**, gli **Endpoint** e i **Processor**.

I **Component** sono in pratica le estensioni di CAMEL che si occupano di fornire connettività con altri sistemi. Un Component fornisce inoltre un interfaccia **Endpoint** riferibile tramite URI. E' quindi possibile inviare o ricevere messaggi da un Endpoint (tramite l'URI che lo identifica) in modo molto semplice. Ad esempio, se volessi ricevere un messaggio da una coda JMS miaCoda ed inviarlo verso una directory del filesystem /tmp perché venga memorizzato, posso usare delle URI così fatte: "jms:miaCoda' e "file:/tmp". La prima parte esplicita il tipo di component in uso e la seconda l'URI a cui si sta facendo riferimento.

I **Processor**, interposti tra due Endpoint, sono usati per manipolare i messaggi che questi Endpoint si scambiano. In pratica gli EIP in CAMEL sono definiti e realizzati tramite l'uso di Processor. Sono supportati più di 40 pattern, estratti dal libro "*Enterprise Integration Patterns*" di Gregor Hohpe e Bobby Woolf.

In breve, gli **Enterprise Integration Patterns**, sono delle **soluzioni a specifici problemi legati al design delle applicazioni**[8]. Sono state pensate nel corso di anni di sviluppi nell'ambito IT e cosa importante, non sono legate a nessuna tecnologia in particolare, quindi a nessun particolare linguaggio o sistema operativo; sono solo dei modelli a cui far riferimento per realizzare un'applicazione efficiente.

Infine, per legare i Processor e gli Endpoint e per poter utilizzare questa tecnologia all'interno delle nostre applicazioni, CAMEL definisce un un DLS, disponibile sia in forma JAVA che XML. Tramite il DLS possiamo specificare le **rotte**, ossia i percorsi che vogliamo far fare al messaggio. Riprendendo l'esempio precedente, scriverei:

- Java DSL

```
from ("jms:miaCoda").to("file:/tmp");
```

- Spring DSL

```
<route>
<from uri="jms:miaCoda" />
<to uri="file:/tmp" />
</route>
```

Nel nostro caso usiamo Camel come **bus interno** a ServiceMix per connettere tra di loro i servizi. E' flessibile e permette di far comunicare tra loro i nostri moduli, direttamente attraverso la JVM oppure in rete distribuendo servizi su macchine diverse, ad esempio con Web Service. Tornando infine a ServiceMix, al giorno d'oggi JBI è diventata una tecnologia ormai obsoleta e l'approccio che questo ESB propone per fare integrazione si basa oramai completamente su Apache Karaf e su Apache Camel; anzi, volendo sintetizzar, si può dire che ServiceMix non è altro che la combinazione di questi due.

2.3 Web Services

Ultima parte di questa presentazione sono i Web Service, una delle possibili realizzazioni del concetto di servizio SOA e probabilmente la più diffusa.

E' un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori; caratteristica fondamentale di un Web Service è quella di offrire un *interfaccia* software (descritta in un formato automaticamente

elaborabile quale, ad esempio, il Web Services Description Language) utilizzando la quale altri sistemi possono interagire con il Web Service stesso attivando le operazioni descritte nell'interfaccia tramite appositi "messaggi" inclusi in una "busta" (la più nota è SOAP): tali messaggi sono, solitamente, trasportati tramite protocollo HTTP e formattati secondo lo standard XML.

Si deduce dalla definizione che le tecnologie alla base dei Web Services sono:

- XML: eXtensible Markup Language
- SOAP: Simple Object Access Protocol
- WSDL: Web Services Description Language
- UDDI: Universal Description, Discovery and Integration

L'uso di XML è stato molto importante per la nascita dei Web Services. E' un linguaggio a marcatori (tag) derivato da SGML, come ad esempio il più conosciuto HTML, ed è utilizzato per memorizzare informazioni in maniera strutturata. XML è un formato indipendente dalle varie piattaforme; ciò è dovuto sia al fatto che è riconosciuto universalmente come standard, ma anche perchè tale tecnologia si basa sul formato testo e quindi un documento XML è leggibile su qualsiasi sistema operativo.

Proprio grazie all'utilizzo di standard basati su XML, tramite un architettura basata su Web Services, applicazioni software scritte in linguaggi di programmazione diversi e implementate su piattaforme differenti possono quindi essere utilizzate, tramite le interfacce che queste espongono pubblicamente e mediante l'utilizzo delle funzioni che sono in grado di effettuare,

per lo scambio di informazioni e l'effettuazione di operazioni complesse sia su reti aziendali come anche su Internet.

2.3.1 SOAP

SOAP è un protocollo leggero per lo scambio di informazioni strutturate in un ambiente distribuito e decentralizzato. E' un set formale di convenzioni che governano il formato e le regole di elaborazione di un messaggio SOAP.[9] Il Simple Object Access Protocol è più una specifica che definisce come deve essere fatto un messaggio per poter essere scambiato tra due applicazioni nell'ambito dei Web Service. Si specificano in particolare:

- struttura del messaggio
- meccanismo di scambio (MEP: Message Exchange Pattern)
- gestione degli errori: specifica il contenuto dei messaggi, detti Fault message, in caso di errore nella comunicazione
- modello di elaborazione dei messaggi: SOAP prevede che un messaggio parta da un mittente e nel tragitto verso il destinatario, possa attraversare nodi intermedi di elaborazione. Il processing model stabilisce come tali nodi intermedi devono comportarsi quando ricevono il messaggio.
- protocol binding: cioè su quale protocollo vengono trasportati i messaggi; le specifiche prevedono HTTP ma non escludono anche l'uso di altri protocolli

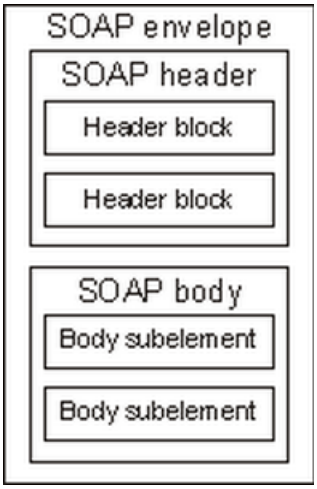


Figura 2.14: Struttura del messaggio SOAP

2.3.1.1 Struttura del messaggio

La struttura esterna è il SOAP Envelope, ossia la busta di cui si parlava in precedenza nella definizione di Web Service. Al suo interno sono presenti:

- il SOAP Header: è opzionale, al suo interno ci possono essere più Header Block (se c'è il SOAP Header allora deve essere presente almeno un Header Block)
- il SOAP Body: è obbligatorio, al suo interno è presente il vero e proprio contenuto del messaggio racchiuso in sotto elementi

Il messaggio SOAP è in formato XML e sebbene la specifica non imponga in che modo devono essere internamente strutturati i messaggi, si segue comunemente il modello RPC. SOAP RPC finisce per essere la definizione di come deve essere strutturato il Body e l'introduzione di un attributo

"Encoding-style" che specifica come deve essere serializzato il messaggio. Per il Body, la regola è:

- nella request, l'unico figlio del Body deve essere il nome del metodo, mentre nella response deve essere lo stesso nome del metodo a cui è stata appesa la stringa "Response"
- i parametri di ingresso e uscita sono i figli dell'unico elemento del Body

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env=
              "http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
  .... Same as call ....
 </env:Header>

 <env:Body>
     <m:chargeReservationResponse
            env:encodingStyle=
                "http://www.w3.org/2003/05/soap-encoding"
            xmlns:m="http://travelcompany.example.org/">
       <m:code>FT35ZBQ</m:code>
       <m:viewAt>
     http://travelcompany.example.org/reservations?code=FT35ZBQ
       </m:viewAt>
     </m:chargeReservationResponse>
 </env:Body>
</env:Envelope>
```

Figura 2.15: Esempio di risposta SOAP

2.3.1.2 Message Exchange Pattern

Definisce sostanzialmente la modalità con la quale i due interlocutori comunicano loro:

- one-way: il chiamante manda il messaggio e non si blocca ma continua le sue operazioni.

- request-response: il chiamante manda il messaggio e si blocca in attesa di risposta

La soluzione prodotta nel seguito, prevede un MEP request-response e l'uso di HTTP come protocollo di trasporto.

2.3.2 WSDL

E' un linguaggio formale in formato XML utilizzato per la creazione di "documenti" per la descrizione del Web Service[10].

Un documento WSDL contiene, relativamente al Web Service descritto, informazioni su:

- cosa può essere utilizzato: cioè le operazioni messe a disposizione dal servizio
- come utilizzarlo: il protocollo di comunicazione per accedere al servizio, il formato dei messaggi accettati in input e restituiti in output
- dove utilizzare il servizio: l'endpoint, cioè l'URI dove localizzarlo.

Esso è composto da cinque elementi:

- Types: definisce i tipi di dati coinvolti nello scambio dei messaggi (definiti secondo XML Schema)
- Messages: specifica le informazioni sui parametri e i loro tipi rispetto ai messaggio di input e output
- PortType: indica le operazioni supportate dal servizio

- Binding: stabilisce il protocollo da utilizzare nello scambio di messaggi
- Service: specifica l'URI alla quale è possibile contattare il servizio

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://thalesgroup.com/ns/service/hello"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
name="HelloServiceService"
targetNamespace="http://thalesgroup.com/ns/service/hello">

<wsdl:types>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://thalesgroup.com/ns/service/hello"
attributeFormDefault="unqualified" elementFormDefault="
    unqualified"
targetNamespace="http://thalesgroup.com/ns/service/hello">
<xs:complexType name="sayHello">
<xs:sequence>
<xs:element minOccurs="0" name="name" type="xs:string"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="sayHelloResponse">
<xs:sequence>
<xs:element minOccurs="0" name="return" type="xs:string"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="makeSum">
<xs:sequence>
<xs:element name="param1" type="xs:double"/>
<xs:element name="param2" type="xs:double"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="makeSumResponse">
<xs:sequence>
<xs:element name="return" type="xs:double"/>
</xs:sequence>
</xs:complexType>
<xs:element name="sayHello" nillable="true" type="sayHello"/>
<xs:element name="sayHelloResponse" nillable="true" type="
    sayHelloResponse"/>
<xs:element name="makeSum" nillable="true" type="makeSum"/>
```

```
<xs:element name="makeSumResponse" nillable="true" type="
    makeSumResponse"/>
</xs:schema>
 </wsdl:types>

<wsdl:message name="makeSum">
    <wsdl:part element="tns:makeSum" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="sayHelloResponse">
    <wsdl:part element="tns:sayHelloResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="sayHello">
    <wsdl:part element="tns:sayHello" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="makeSumResponse">
    <wsdl:part element="tns:makeSumResponse" name="parameters">
    </wsdl:part>
</wsdl:message>

<wsdl:portType name="HelloService">
    <wsdl:operation name="sayHello">
      <wsdl:input message="tns:sayHello" name="sayHello">
    </wsdl:input>
      <wsdl:output message="tns:sayHelloResponse" name="
    sayHelloResponse">
    </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="makeSum">
      <wsdl:input message="tns:makeSum" name="makeSum">
    </wsdl:input>
      <wsdl:output message="tns:makeSumResponse" name="
    makeSumResponse">
    </wsdl:output>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="HelloServiceServiceSoapBinding" type="
    tns:HelloService">
    <soap:binding style="document" transport="http://schemas.
    xmlsoap.org/soap/http"/>
    <wsdl:operation name="sayHello">
      <soap:operation soapAction="urn:#sayHello" style="document
```

```
    "/>
79      <wsdl:input name="sayHello">
80        <soap:body use="literal"/>
81      </wsdl:input>
82      <wsdl:output name="sayHelloResponse">
83        <soap:body use="literal"/>
84      </wsdl:output>
85    </wsdl:operation>
86    <wsdl:operation name="makeSum">
87      <soap:operation soapAction="urn:#makeSum" style="document"
    />
88      <wsdl:input name="makeSum">
89        <soap:body use="literal"/>
90      </wsdl:input>
91      <wsdl:output name="makeSumResponse">
92        <soap:body use="literal"/>
93      </wsdl:output>
94    </wsdl:operation>
95</wsdl:binding>
96
97<wsdl:service name="HelloServiceService">
98    <wsdl:port binding="tns:HelloServiceServiceSoapBinding" name
    ="HelloServicePort">
99      <soap:address location="http://localhost:8181/cxf/
    helloCamelService"/>
100   </wsdl:port>
101</wsdl:service>
102
103</wsdl:definitions>
```

Listing 2.1: Esempio di documento WSDL

2.3.3 UDDI

E' l'acronimo di Universal Description, Discovery and Integration ed è un registry, ossia una base di dati ordinata e indicizzata, basato su XML e indipendente dalla piattaforma hardware, che permette alle aziende la pubblicazione dei propri servizi (fornendo la descrizione WSDL), cioè quelli che offrono verso l'esterno, e ai consumatori di trovare tali servizi e usufruirne[11].

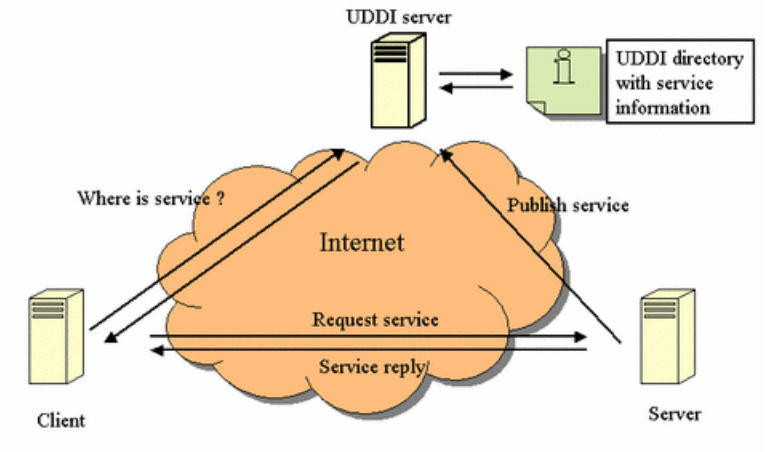


Figura 2.16: Esempio di utilizzo del registro UDDI

La informazioni memorizzate in un registry UDDI sono organizzate in modo da permettere diverse forme di ricerca, in particolare sono ordinate secondo tre cataloghi:

- White Pages: contengono informazioni generiche sull'azienda che ha pubblicato il servizio.
- Yellow Pages: contiene una classificazione tassonomica dei servizi per effettuare ricerche in base alla categoria di appartenenza dello stesso
- Green Pages: contiene informazioni tecniche sui servizi grazie alle quali gli stessi possono essere invocati

Si precisa che la soluzione impiegata nell'ambito di questo lavoro non richiederà l'uso di un UDDI, in quanto il WSDL del servizio a cui accederemo sarà recuperabile ad una URI nota, fornita dall'ESB stesso.

2.3.4 Alcune API

Considerando l'interoperabilità che i Web Service forniscono, sono stati approcciati due linguaggi diversi, per lo sviluppo del servizio (Java) e dei consumatori del servizio (Java e C++). Sono state usate le seguenti librerie:

- Java: facendo riferimento alle note API JAX-WS (Java API for XML Web Services), è stata utilizzata l'implementazione fornita dal framework Apache CXF.
- C++: si è fatto uso di gSOAP, un toolkit di sviluppo per SOAP/XML Web services. Il toolkit analizza i documenti WDSL e gli XML schema associati mappandoli in strutture dati che permettono di invocare facilmente un servizio all'interno di un listato C/C++.

Capitolo 3

Analisi dello stato dell'arte dei sistemi di sicurezza per middleware distributi

3.1 Descrizione contesto di Sicurezza e Requisiti di progetto

Partendo dalla conoscenza dell'architettura e delle tecnologie alla base del progetto PITAGORA, il lavoro di tesi si è principalmente focalizzato sulla messa in sicurezza di una porzione di infrastruttura. Tale porzione riguarda esattamente il canale di comunicazione tra un "modulo partner", ossia un'applicazione di un partner PITAGORA, e la Piattaforma stessa. La Figura 3.1 mostra una rappresentazione dell'architettura semplificata al fine di individuare le componenti interessate e il problema su cui si è operato. Una

prima osservazione consente di notare la distinzione tra dominio dell'**owner** (in questo caso THALES) e dominio del **partner**. All'interno del dominio dell'owner ricadono l'ESB, varie applicazioni di competenza dell'owner e l'HMI (Human Machine Interface), quest'ultimo realizzato come Application Server e anch'esso assimilabile ad un entità che intende comunicare con l'ESB. Esternamente identifichiamo le applicazioni dei partner che, integrate alla piattaforma necessitano di comunicare con essa.

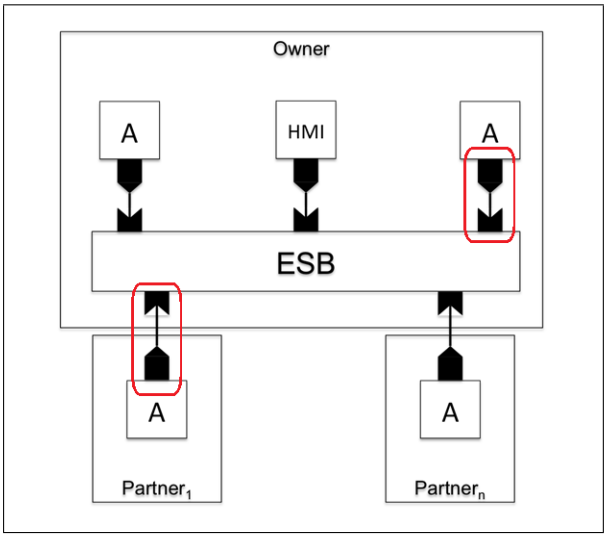


Figura 3.1: Visione dell'architettura e porzione di interesse

L'area contornata in rosso rappresenta invece il canale da mettere in sicurezza, per ogni applicazione afferente alla Piattaforma; per fare ciò sono stati seguiti i seguenti step:

1. sono state inizialmente analizzate due soluzioni sulla base dello stato dell'arte;

2. è stato valutato l'uso di Certification Authority, ricercata e selezionata la più appropriata;

3. realizzato un servizio presso l'ESB ed esposto come Web Service;

4. realizzati dei connettori (Java, C++) che interagiscono con il servizio;

5. configurato il servizio web e i connettori per una comunicazione in sicurezza;

Non è stata richiesta una precisa logica di business, né tanto meno complessa, quindi l'interfaccia del Web Service fornisce una serie di funzioni basilari per lo scambio di dati secondo un MEP request-response.

I connettori, rappresentati in Figura 3.1 con una forma simile ad una freccia, sono utilizzati come giunzione tra il modulo di un partner e la Piattaforma. Il connettore ha in sintesi due funzioni:

1. Fornire una rappresentazione del servizio e quindi ricavare l'interfaccia per comunicare con esso (in modalità sicura).

2. Mappare il modello dei dati del modulo del partner nel modello dati utilizzato dalla piattaforma.

3.2 Requisiti di sicurezza

Con riferimento alla Figura 3.1, la piattaforma Pitagora è quindi composta da una serie di applicazioni interconnesse per mezzo di un Enterprise Service Bus. L'ESB risiede all'interno del dominio dell'owner della piattaforma (THALES), mentre le applicazioni possono appartenere allo stesso dominio o

a domini diversi, cioè quelli dei rispettivi partner. La partnership è variabile nel tempo ossia, nuovi partner possono entrare a far parte della piattaforma oppure gli attuali partner possono abbandonarla.

La piattaforma, trovandosi in un contesto di infrastruttura critica, deve garantire il requisito di **sicurezza**. Intuitivamente, ciò significa che la piattaforma sia in grado di garantire *accuratezza*, *confidenzialità* e *autenticità* dell'informazione.

- **Accuratezza** significa che il dato non venga modificato in modo non autorizzato o non rilevabile. L'accuratezza si ottiene garantendo l'*integrità* del dato.

- La **confidenzialità** garantisce che i dati scambiati con la piattaforma siano accessibili solo ai legittimi utenti/applicazioni. Una violazione con successo di tale requisito è detta rivelazione. Si noti che la confidenzialità può inoltre rendersi necessaria al fine di prevenire il caso in cui un ipotetico attaccante acquisisca dati a sufficienza per un successivo attacco che vada a colpire l'integrità o la disponibilità del sistema.

- L'**autenticità** è intesa come il processo di verifica dell'identità del servizio che fa richiesta. Un identità può far riferimento ad un utente, un applicazione o ad un servizio esterno che fa richiesta. Per quanto riguarda l'ambito SOA, ciò significa sapere chi sta contattando il servizio.

In aggiunta e, al pari rilevante, è stato considerato l'aspetto dell' **autorizzazione**. Il concetto di autenticazione precede logicamente la nozione di Controllo Accessi. Il Controllo Accessi previene l' "usurpazione", intesa come

l'uso delle funzionalità o servizi del sistema da parte di entità non autorizzate. Precisamente, il controllo accessi impedisce a degli estranei (entità non autorizzate) di ottenere l'accesso al sistema, imponendo e facendo rispettare determinate restrizioni relative a cosa un membro (entità autenticata) può fare.

3.3 Sistemi a chiave pubblica e sistemi a chiave privata: un confronto

La crittografia è la scienza che si occupa di proteggere delle informazioni rendendole incomprensibili a chi le dovesse intercettare, ossia le rende leggibili solo al corretto destinatario. La crittografia moderna segue il principio di Kerckhoffs, il quale afferma che l'algoritmo deve essere pubblico ma basato su una trasformazione parametrica, il cui parametro è detto chiave e deve essere mantenuto segreto. Il problema principale della crittografia è la gestione delle chiavi, ovvero la regola da seguire per decifrare il messaggio. Possiamo distinguere due tipi di approcci:

- **simmetrico**, o a chiave privata
- **asimmetrico**, o a chiave pubblica

3.3.1 Crittografia simmetrica

Il sistema a chiave simmetrica fa uso di un'unica chiave, utilizzata per codificare e decodificare il messaggio che viene scambiato tra due interlocutori. La

chiave deve essere nota sia al mittente che al destinatario, i quali dovranno scambiarsela in anticipo.

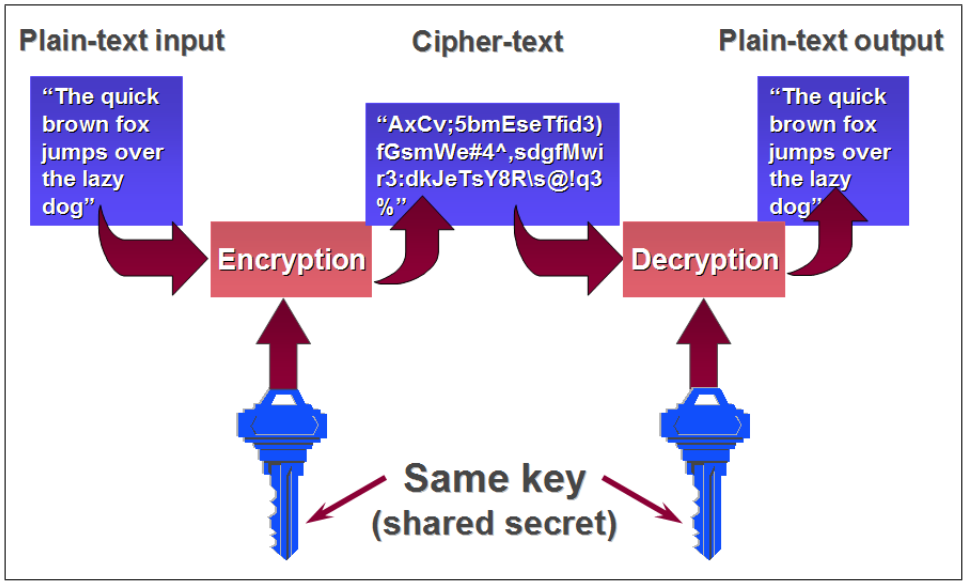


Figura 3.2: Schema a crittografia simmetrica

Gli algoritmi simmetrici si basano su trasposizioni e sostituzioni di elementi e si suddividono in cifrari a "blocco", che codificano e decodificano il testo a blocchi di bit, e cifrari a "flusso" che operano bit a bit. Tra i più noti algoritmi si possono rammentare: DES, 3DES e AES. Una caratteristica importante è che gli algoritmi a chiave simmetrica sono molto efficienti in termini computazionali e questo li rende ideali per cifrare messaggi in tempo reale. Si noti che, per la comunicazione con più parti, bisogna usare una chiave segreta per ognuno degli interlocutori. Il problema che si pone in presenza di questi algoritmi, dato che la chiave deve essere nota ad entrambi gli interlocutori, è la *modalità con la quale tale chiave viene condivisa*. Una possibile

soluzione per stabilire una chiave è sfruttare il protocollo Diffie-Helmann ma non è sicuro in quanto soggetto a Man-in-the-Middle attack. Valida alternativa risiede nella crittografia asimmetrica, di cui si danno alcuni accenni nel seguito, abbinata all'uso di un'autorità di certificazione.

3.3.2 Crittografia asimmetrica

La crittografia asimmetrica, o a chiave pubblica, è invece caratterizzata da una coppia di chiavi: la prima, detta **chiave pubblica** è nota a chiunque, la seconda detta **chiave privata** è nota solo al suo titolare.

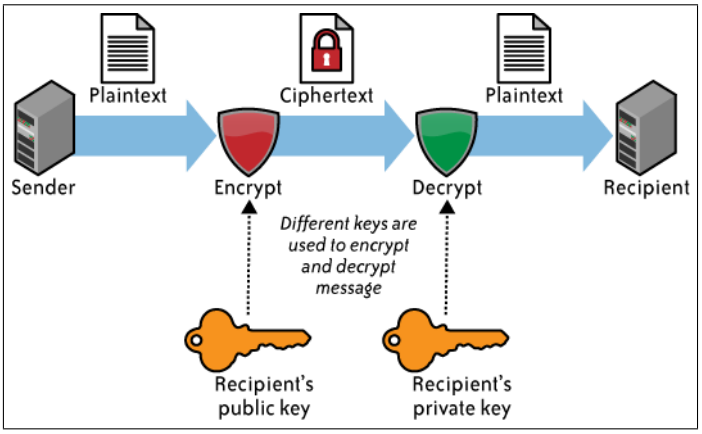


Figura 3.3: Schema a crittografia asimmetrica

Il funzionamento, come da Figura 3.3, prevede che un Sender cifri un messaggio con la chiave pubblica del Recipient e quest'ultimo possa decifrare il messaggio utilizzando la propria chiave privata. Da un punto di vista formale, gli algoritmi associati a questa categoria, basano la loro robustezza sulla complessità di un problema NP ed inoltre la sola conoscenza della chiave pubblica non rende possibile dedurre la chiave privata.

Lo schema asimmetrico permette di garantire i tre requisiti fondamentali: confidenzialità, integrità e autenticazione. Al fine di garantire la riservatezza e integrità del messaggio, un mittente cifra tale messaggio con la chiave pubblica del destinatario e quest'ultimo decifra tramite la sua chiave privata, accertandosi così che esso non sia stato alterato; infatti se il messaggio fosse stato alterato verrebbe meno la corrispondenza tra chiave pubblica e chiave privata e il messaggio non si decifrerebbe. In questo modo garantisco la segretezza ma non la paternità del messaggio, infatti chiunque potrebbe usare la chiave pubblica del destinatario e inviargli un messaggio cifrato.

Per ottenere l'autenticazione invece è possibile sfruttare la **firma digitale**, invertendo l'uso delle chiavi rispetto al caso precedente. Il mittente cifra il messaggio con la sua chiave privata, il destinatario lo decifra con la chiave pubblica del mittente, verificandone l'autenticità. Se la verifica ha esito positivo è assicurata l'autenticità del documento poiché solo il mittente può aver usato la chiave privata per la cifratura; in caso contrario il documento non appartiene al mittente.

Apponendo la sola firma digitale però, chiunque potrebbe decifrare il messaggio utilizzando la chiave pubblica del mittente. Per aggiungere il requisito di sicurezza quindi si potrebbe operare usando un messaggio cifrato e firmato; il mittente firma il messaggio con la sua chiave privata, poi cifra il tutto con la chiave pubblica del destinatario; il destinatario decifra il messaggio con la sua chiave privata e verifica la firma con la chiave pubblica del mittente.

L'uso di algoritmi a chiave asimmetrica, a differenza di quelli simmetri-

ci, risulta inefficiente, specialmente se il messaggio da crittografare è molto grande. Di fatto si procede sfruttando lo schema asimmetrico per instaurare una chiave di sessione tra gli interlocutori e poi sfruttare la crittografia simmetrica.

Come Diffie-Hellman, anche questo schema soffre del MiM attack (vedi Figura 3.4):

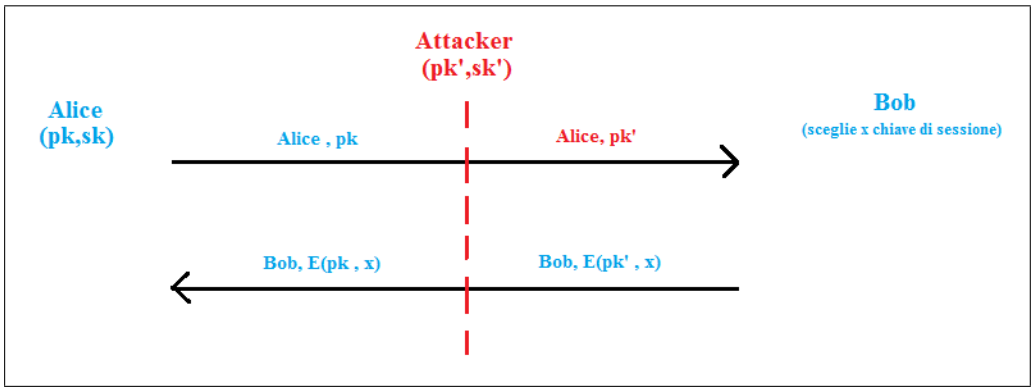


Figura 3.4: Semplice esempio di Man-in-the-middle attack

La soluzione ad un attacco di questo tipo è utilizzare, una terza parte fidata che certifichi la chiave pubblica, ossia leghi l'identità di un soggetto alla sua chiave pubblica. Si rende quindi necessaria una Certification Authority (CA).

Dopo questa breve panoramica relativa alla crittografia simmetrica e asimmetrica, ci siamo concentrati nell'individuare soluzioni standard che si basano su tali concetti, per fornire servizi di sicurezza nello scambio dati in rete.

3.4 Analisi dello stato dell'arte per implementazioni a chiave pubblica in sistemi SOA

Dovendo garantire una comunicazione sicura nell'ambito di tecnologie basate su Web Service, sono state considerate due possibili alternative. Entrambe garantiscono il raggiungimento dei requisiti previsti di cui si è precedentemente discusso, ma concettualmente differiscono per il livello in cui viene effettivamente attuata la sicurezza dei dati[12]; si distingue in:

- *Sicurezza a livello di trasporto*: La sicurezza dei dati viene garantita dal protocollo di trasporto scelto per la comunicazione tra due nodi. Basato tipicamente sulla combinazione di SSL/TLS ed HTTP, quindi comunicazione via HTTPS. Tale soluzione prevede la messa in sicurezza del canale e quindi dell'intero messaggio, dal sender fino all'endpoint del servizio. E' una soluzione molto diffusa, utile se la comunicazione è diretta (point-to point) e meno complessa da realizzare rispetto ad altri approcci.

- *Sicurezza a livello di messaggio*: rappresenta una forma di sicurezza applicativa. Nella comunicazioni basate su messaggi SOAP, le informazioni di sicurezza sono contenute come header dei messaggi, all'interno di particolari tag. E' una soluzione spesso adottata in presenza di comunicazioni in cui il percorso dei messaggi prevede di attraversare più nodi intermedi connessi con protocolli differenti. E' anche utile laddove si decida di mettere in sicurezza (firmare e cifrare) solo parti del messaggio, così che eventuali nodi intermedi possano comunque vedere

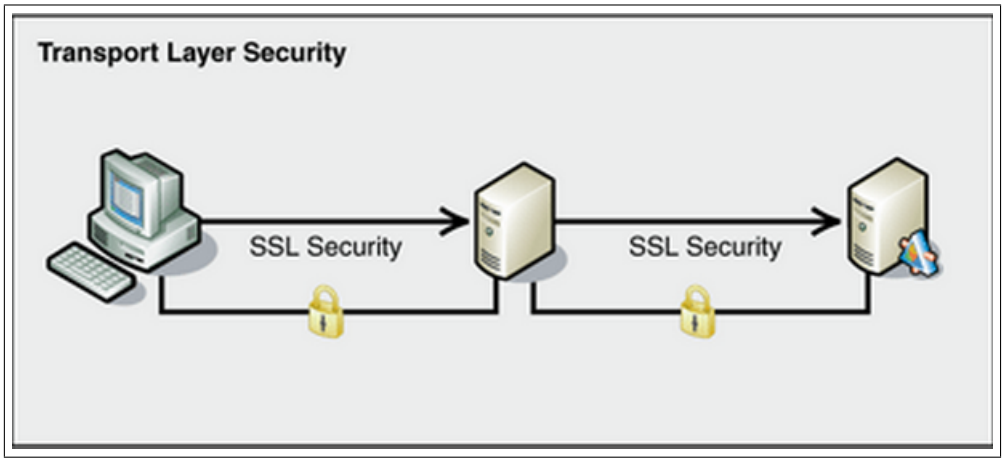


Figura 3.5: Sicurezza a livello di trasporto

le parti del messaggio non sensibili. Per questa tipologia di sicurezza è stata presa in considerazione la specifica WS-Security.

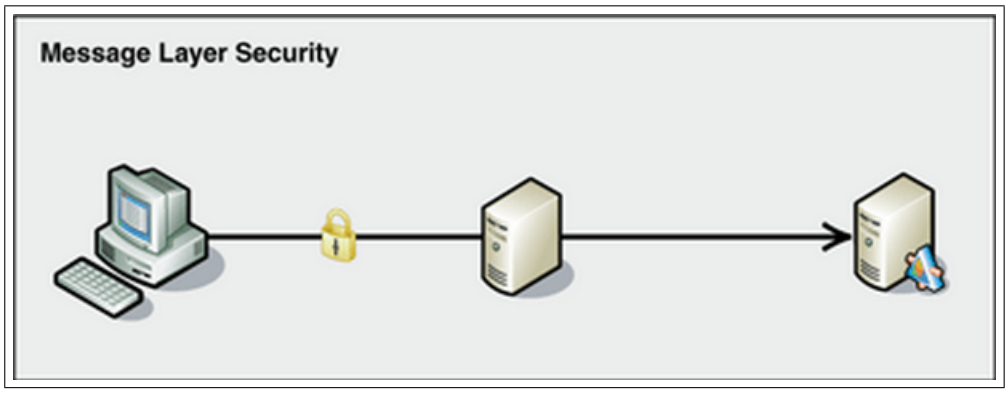


Figura 3.6: Sicurezza a livello di messaggio

3.4.1 SSL/TLS

Transport Layer Security (TLS)[13] e il suo predecessore **Secure Sockets Layer (SSL)**, nel campo delle telecomunicazioni e dell'informatica, sono dei protocolli crittografici (ovvero una suite) che permettono una comunicazione sicura dal sorgente al destinatario su reti TCP/IP (come ad esempio Internet) fornendo autenticazione, integrità dei dati e cifratura, operando al di sopra del livello di trasporto[14].

3.4.1.1 La suite

La suite, come mostra la Figura 3.7 si compone dei seguenti protocolli:

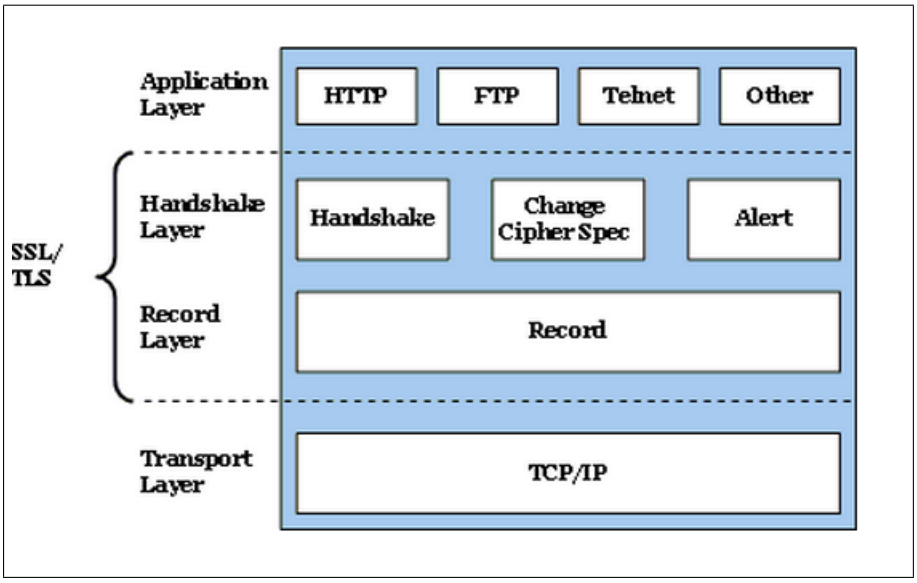


Figura 3.7: Composizione di SSL e posizione nello stack

- **Handshake**: permette di contrattare i parametri relativi al meccanismo di sicurezza;

- **ChangeCipherSpec**: : Innesca il cambio di cifrario usato, in particolare determina l'inizio delle comunicazioni cifrate una volta stabilite le chiavi.
- **Alert**: Invia segnalazioni di errori e di situazioni di warning;
- **Record**: Permette lo scambio di record come unità elementare di trasmissione, effettuando la cifratura e apponendo il MAC per garantire l'integrità del record.

3.4.1.2 Sessione e connessione

SSL/TLS distingue tra **sessione** e **connessione**. Una sessione crea un'associazione logica tra un client e un server; ha luogo una volta sola all'inizio della comunicazione per mezzo del protocollo di Handshake e permette di stabilire un segreto condiviso tra i due interlocutori. All'interno della sessione vengono poi create le connessioni; ognuna di esse ha le proprie chiavi (cifratura e MAC) create a partire dal segreto stabilito in sessione. Graficamente, la Figura 3.8 , mostra come può essere immaginata tale distinzione.

Figura 3.8: Differenza tra sessione e connessione SSL/TLS

Al termine del protocollo di Handshake, se tutto è andato a buon fine e la sessione è instaurata, i due partecipanti avranno lo stesso SESSION STATE con le seguenti informazioni:

1. Session Id
2. Peer Certificate (X.509v3)
3. Compression Method
4. Cipher Suite
5. Pre master secret (segreto condiviso)

L'obiettivo della sessione è ottenere uno stato comune e soprattutto condividere un segreto comune, il pre master secret. Facendo uso di crittografia asimmetrica, risulta computazionalmente onerosa, dunque viene eseguita la prima volta e poi per ogni successiva comunicazione si usano le connessioni, all'interno delle quali, come si vedrà, una volta stabilite delle chiavi si utilizzerà esclusivamente crittografia simmetrica, molto più rapida. L'obiettivo della creazione di una connessione è far si che client e server abbiano lo stesso CONNECTION STATE, così caratterizzato:

1. Server Nonce
2. Client Nonce
3. Server Write MAC
4. Client Write MAC
5. Server Write key

6. Client Write key

7. Initialization Vector

8. Sequence Number

Le chiavi sono generate a partire dal Pre master secret, deterministicamente ambo i lati, senza che client e server debbano comunicare. Inoltre queste chiavi sono diverse per client e server, sia quelle per la cifratura che per il MAC.

3.4.1.3 Protocollo di Record

Il Protocollo Record, una volta terminato l'Handshake, all'atto delle connessioni, incapsula i dati spediti dai livelli superiori assicurando la confidenzialità e l'integrità della comunicazione.

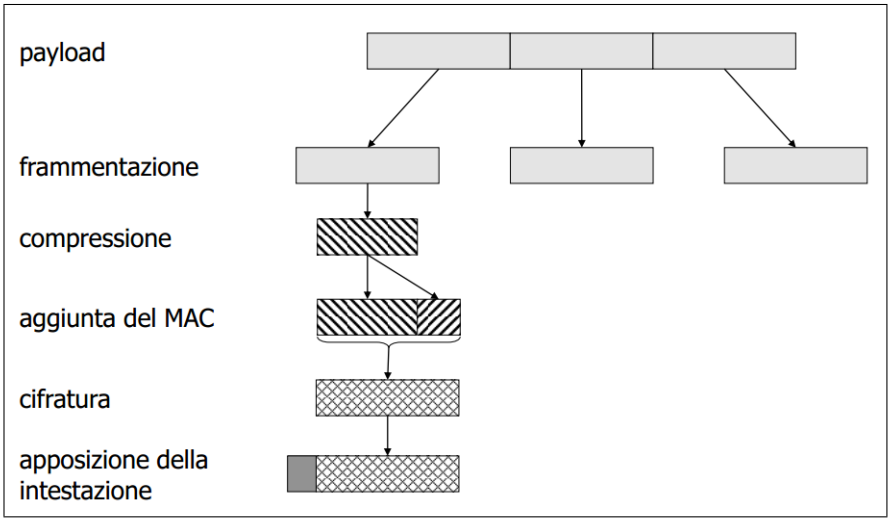


Figura 3.9: Fasi del protocollo di Record

- La **frammentazione** frammenta i dati applicativi in blocchi di al più 2^{14} byte.
- La **compressione** deve essere senza perdita e non deve far aumentare le dimensioni di un blocco di più di 1024 byte.
- Il **MAC** utilizza il [Server|Client] write MAC secret, il sequence number, il blocco compresso, pad, ...
- La **cifratura** utilizza la [Server|Client] write key, può essere a blocchi o a caratteri e non deve far aumentare le dimensioni di un blocco di più di 1024 byte.

3.4.1.4 Il protocollo di Handshake

Il protocollo di Handshake (vedi Figura 3.10) permette di stabilire una sessione sicura, cioè permette:

- al **client** e al **server** di autenticarsi a vicenda;
- di negoziare la **suite di cifratura (chiper suite)**:
  - il metodo per lo scambio delle chiavi;
  - l'algoritmo di cifratura (utilizzato nel protocollo di Record);
  - l'algoritmo per il MAC (utilizzato nel protocollo di Record);
- di stabilire un segreto condiviso (master secret)

Il contenuto dei vari messaggi è il seguente:

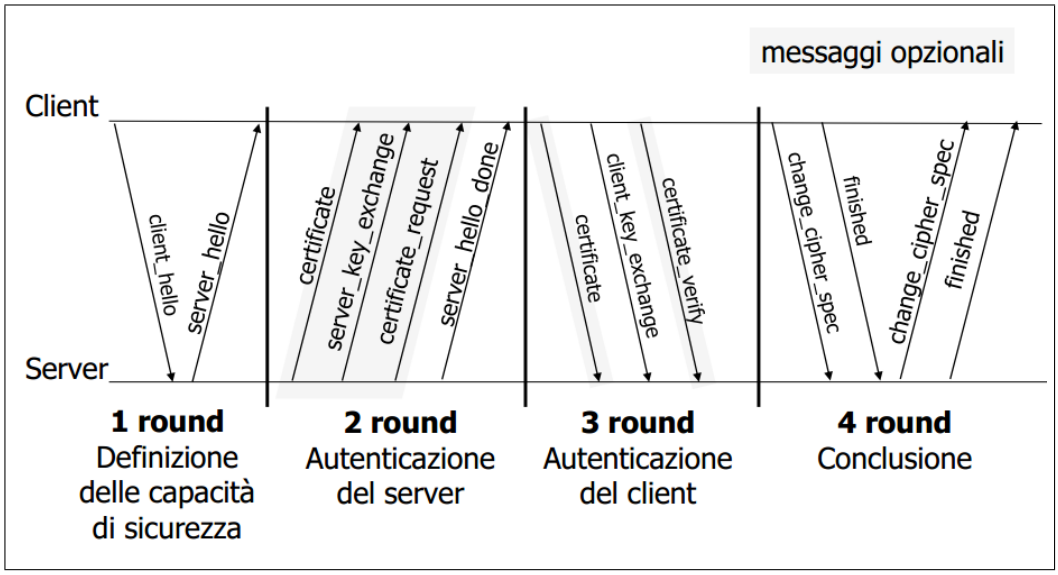


Figura 3.10: Fasi del protocollo di Handshake con mutua autenticazione

- **client hello**: SSL version, Random (client timestamp + random byte), session ID, cipher suite, compression method
- **server hello**: SSL version, Random (server timestamp + random byte), session ID, cipher suite, compression method
- **certificate**: certificato X.509v3 del server
- **server key exchange**: se le parti concordano nell'uso di Diffie-Helman come algoritmo di "key establishment", il server invia il suo contributo al client
- **certificate request**: tipo, autorità di certificazione
- **server hello done**: nessun parametro (è più un segnalatore)
- **certificate**: certificato X.509v3 del client

- **client key exchange**: contiene il pre master secret, generato come valore random e cifrato con la chiave pubblica del server contenuta nel certificato

- **certificate verify**: contiene l'hash di server nonce, client nonce, pre master secret, Server Certificate, Client Certificate (permette al server di sapere che il client possiede la chiave privata legata al certificato che gli ha inviato)

- **change chiper spec**: (messaggio in chiaro che ha l'obiettivo di rendere operativa la chiper suite negoziata)

- **finished**: contiene l'hash ottenuto con la chiave MAC di connessione di (client nonce, server nonce, pre master secret, Server Certificate), tutto cifrato con la chiave di connessione del client

- **change chiper spec**: (messaggio in chiaro che ha l'obiettivo di rendere operativa la chiper suite negoziata)

- **finished**: contiene l'hash ottenuto con la chiave MAC di connessione di (client nonce, server nonce, pre master secret, Client Certificate), tutto cifrato con la chiave di connessione del server

Alla fine dell'Handshake, con i messaggi "finished", se entrambe le parti riescono a decifrare il messaggio allora credono di avere rispettivamente le giuste chiavi. Si noti che, all'atto della connessioni, il server tramite il Session ID controlla se esiste una sessione già instaurata con il client ed in tal caso esegue una versione ridotta del protocollo di Handshake. Ultima precisazione, riguarda il modo in cui client e server generano le chiavi per cifratura e MAC

a partire dal pre master secret. La Figura 3.11 mostra come tale processo avviene.

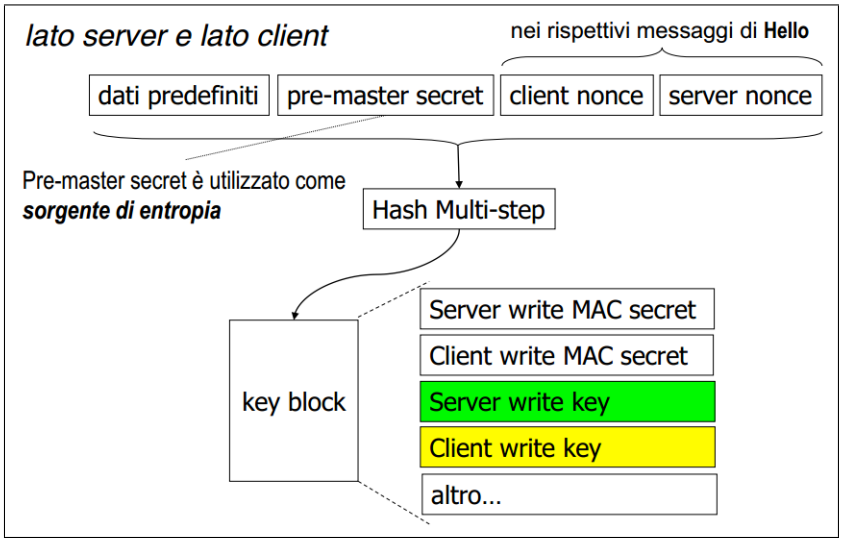


Figura 3.11: Generazione delle chiavi

3.4.2 WS-Security

La specifica WS-Security definisce un'estensione di SOAP che implementa autenticazione, integrità e confidenzialità a livello messaggio[15]. L'obiettivo di questa specifica non è introdurre nuove tecniche, ma quello di utilizzare le soluzioni esistenti in materia di sicurezza delle comunicazioni con SOAP ed i Web Service.

Il punto di ingresso di WS-Security è un header SOAP, chiamato <Security>. Contiene i dati riguardanti la sicurezza e le informazioni necessarie per implementare meccanismi come firma e cifratura. Questo elemento può essere presente più volte all'interno del messaggio se le informazioni di sicurezza da

inserire sono destinate a differenti destinatari (indicati con l'attributo actor). Due header <Security> non possono avere il solito actor, mentre un header senza actor specificato, può essere consumato da qualsiasi ricevente.

Vediamo un esempio di messaggio SOAP con un header <Security>:

```
<SOAP:Envelope xmlns:SOAP="...">
  <SOAP:Header>
    <wsse:Security SOAP:actor="..." SOAP:mustUnderstand="...">
      ...
    </wsse:Security>
  </SOAP:Header>
  <SOAP:Body Id="MsgBody">
      <!-- SOAP Body data -->
  </SOAP:Body>
</SOAP:Envelope>
```

3.4.2.1 Autenticazione

Uno dei requisiti centrali di un'infrastruttura di comunicazione è la possibilità di fornire e ricevere referenze attendibili sull'identità dei soggetti coinvolti nella comunicazione. Queste referenze sono informazioni aggiuntive che trovano naturale collocazione nell'header SOAP del messaggio.

Ci sono molteplici modi per fornire prove della propria identità e WS-Security fornisce un metodo astratto per implementarle. I metodi che analizzeremo sono:

- Username/Password
- Certificati X.509 su PKI

Vediamo ad esempio di autenticazione tramite Username e Password utilizzata anche in HTTP. Le informazioni sono aggiunte all'Header SOAP tramite l'elemento UsernameToken:

```
<!-- No Password -->
<UsernameToken>
  <Username>Alice</Username>
</UsernameToken>

<!-- Clear Text Password -->
<UsernameToken>
  <Username>Alice</Username>
  <Password Type="wsse:PasswordText">Wonderland</Password>
</UsernameToken>

<!-- Digest: SHA1 hash of base64-encoded Password -->
<UsernameToken>
  <Username>Alice</Username>
  <Password Type="wsse:PasswordDigest">
    gpBDXjx79eutcXdtlULIlcrSiRs=<Password>
  <Nonce>h52sI9pKV0BVRPUolQC7Cg==</Nonce>
  <Created>2002-11-04T19:16:50Z</Created>
</UsernameToken>
```

Nel primo caso inviamo il nome dell'utente senza nessuna password di sicurezza. Questo tipo di autenticazione deve essere utilizzato in combinazione con altre tecniche di sicurezza perché chiaramente poco robusta.

Stesse considerazioni anche per la seconda tecnica di autenticazione, che prevede di passare la password di sicurezza in chiaro nel messaggio.

Fornisce invece una buon livello di sicurezza il terzo tipo di autenticazione, che invia la password di sicurezza codificata ed accompagnata da un digest (o impronta) in combinazione con la data di creazione per ovviare ad attacchi di tipo replay.

Un'altra opzione è quella di inviare un certificato X.509 che, avvalendosi dell'infrastruttura a chiave pubblica, fornisce l'identità di un soggetto e garantisce l'attendibilità di tali informazioni.

Il certificato X.509 viene incluso in un messaggio in un elemento WS-Security chiamato BinarySecurityToken. L'algoritmo usato per la codifica

viene specificato nell'attributo EncodingType mentre il tipo di certificato è specificato in ValueType.

```
<wsse:BinarySecurityToken
    ValueType="wsse:X509v3"
    EncodingType="wsse:Base64Binary"
    Id="...">MIIHdjCCB...</wsse:BinarySecurityToken>
```

3.4.2.2 Firma

Il processo di autenticazione ci fornisce garanzie sull'identità del soggetto con cui stiamo scambiando informazioni. Quello che non sappiamo è se le informazioni che giungono a destinazione siano le stesse inserite dal mittente e che non abbiano subito alterazioni durante il tragitto. Non abbiamo quindi garanzie sull'integrità dei dati, garanzie che possiamo fornire firmandoli.

WS-Security si appoggia alla specifica **XML Signature** per firmare un messaggio[16]. Una volta che un messaggio è stato firmato, è praticamente impossibile poterlo modificare senza che il destinatario non se ne accorga. La firma non impedisce che il messaggio sia letto, ma assicura al ricevente che

- Le parti firmate del messaggio non sono state modificate dopo la firma
- Il soggetto che ha apposto la firma è lo stesso identificato dal certificato.

A grandi linee, quando viene firmata una parte del messaggio, viene aggiunto un ID alla parte in chiaro nel messaggio, il certificato del firmatario, anch'esso identificabile univocamente, e un elemento Signature contenente l'elemento firmato, i riferimenti al certificato del firmatario e all'elemento in chiaro oltre ai dettagli del processo di firma.

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod Algorithm="...xml-exc-c14n#"/>
    <SignatureMethod Algorithm="...xmldsig#rsa-sha1"/>
    <Reference URI="#myBody">
        ...
      <DigestMethod Algorithm="...xmldsig#sha1"/>
      <DigestValue>EULddytSo1...</ds:DigestValue>
    <Reference>
  <SignedInfo>
  <SignatureValue>
   BL8jdfToEb1l/vXcMZNNjPOV...
  <SignatureValue>
  <KeyInfo>
    <SecurityTokenReference>
      <Reference URI="#MyX509Token"/>
    </SecurityTokenReference>
  </KeyInfo>
</Signature>
```

Nella prima parte del messaggio troviamo il <CanonicalizationMethod>. Qualsiasi documento che viene firmato deve esser prima portato in forma canonica. Per la natura dell'XML, non esiste un modo di definire l'ordine degli attributi, ne di come trattare gli spazi. Il processo di canonizzazione elimina gli spazi bianchi e ordina gli attributi secondo uno specifico schema.

Il valore della firma è contenuto nell'elemento SignatureValue, SecurityTokenReference riferisce il certificato del firmatario, mentre Reference contiene un riferimento all'elemento firmato.

In questo modo è possibile firmare parti diverse con certificati diversi, risolvendo i problemi addizionali inerenti alle comunicazioni end-to-end.

3.4.2.3 Cifratura

Abbiamo adesso gli strumenti e le tecniche per garantire integrità e autenticità del contenuto delle informazioni scambiate. Dobbiamo ancora forni-

re garanzie di confidenzialità di tali informazioni, ovvero impedire che dati sensibili possano essere letti da soggetti non autorizzati.

Quello che vogliamo è la possibilità di cifrare il contenuto del messaggio, o parti di esso, di modo che solo il destinatario sia in grado di decifrarlo e leggerlo. Anche in questo caso il WS-Security si appoggia su uno standard preesistente e collaudato, l'**XML Encryption**[17].

Quando si codificano i dati, si può scegliere la codifica simmetrica o asimmetrica. La prima richiede di condividere un'informazione segreta. Infatti la chiave usata per cifrare è la medesima usata per decifrare. Questa soluzione è efficace se si ha un buon controllo sulle parti in causa e un buon livello di fiducia su chi detiene le chiavi, ma pone il problema su come distribuire le chiavi.

Se invece vogliamo un metodo che non pone il problema della distribuzione delle chiavi possiamo utilizzare la codifica asimmetrica che abbiamo usato anche per la firma. Mentre nella firma usavamo la chiave privata per firmare e il certificato pubblico per verificare l'integrità, adesso usiamo il certificato pubblico per cifrare e la chiave privata per riportare le parti cifrate in chiaro.

```
<?xml version='1.0'?>
<EncryptedData
  xmlns="http://www.w3.org/2001/04/xmlenc#"
  MimeType="text/xml">
  <EncryptionMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmlsig#">
    <ds:KeyName>MyKeyIdentifier</ds:KeyName>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>B457V645B45.....</CipherValue>
  </CipherData>
</EncryptedData>
```

3.5 Analisi e studio dello stato dell'arte di sistemi per distribuzione chiavi

3.5.1 PKI: Public Key Infrastructure

La soluzione per garantire l'autenticità delle chiavi pubbliche risiede nel concetto di infrastruttura a chiave pubblica (*Public Key Infrastructure*)[18]. L'idea è di istituire una terza parte fidata, detta Autorità di Certificazione (*Certification Authority*, CA) che si occupi di certificare l'autenticità delle chiavi pubbliche degli utenti.

Il *certificato* è una struttura dati che lega l'identificatore di un soggetto alla sua chiave pubblica, facendo uso della firma digitale dell'autorità di certificazione.

3.5.1.1 Certification Authority e Certificati

Consideriamo il caso più semplice, ossia il modello con una singola CA come illustrato in Figura 3.12.

Un *security domain* è un (sotto-)sistema sotto il controllo di una singola autorità di cui tutte le altre entità si fidano. Il *certificate directory* è un database accessibile in sola lettura che memorizza certificati e gestito dalla terza parte fidata.

La CA, per **rilasciare un certificato**, svolge due importanti compiti:

1. verifica l'identità del soggetto (tipicamente tramite procedure non crittografiche)

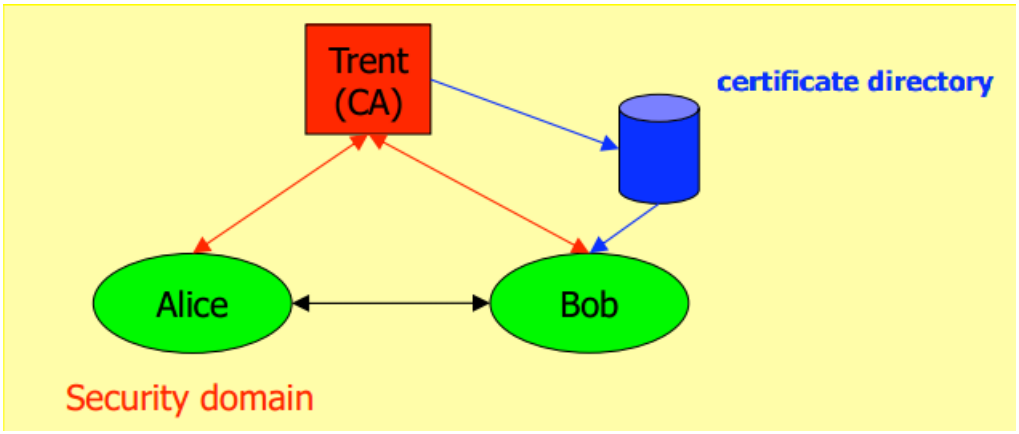


Figura 3.12: Modello a singola Certification Authority

2. verifica l'autenticità della chiave da certificare. In questo caso si considerano due possibili scenari:

   - *Scenario 1*: La coppia di chiavi pubblica-privata è generata dalla CA e trasferita al soggetto in modo da preservarne l'autenticità e la segretezza.
   - *Scenario 2*: La coppia di chiavi pubblica e privata è generata dal soggetto e la chiave pubblica è trasferita alla CA in modo da preservarne l'autenticità. In questo caso la CA richiede al soggetto di fornire una prova dell'effettivo possesso della corrispondente chiave privata (*challenge-response*)

Una volta identificato il soggetto e verificata l'appartenenza della chiave pubblica si può generare il certificato. La struttura minima di un certificato (una visione completa viene data nel seguito) è la seguente:

$$C(T,A) = \text{Alice}, pk_A, L, S(sk_T, \text{Alice}||pk_A||L)$$

I singoli componenti rappresentano:

- Alice: l'identificativo del soggetto
- pk_A: la chiave pubblica di Alice
- L: periodo di validità del certificato
- $S(sk_T, Alice||pk_A||L)$: firma apposta dalla CA, per mezzo della sua chiave privata. E' per mezzo di questa firma che si riesce a legare Alice alla sua chiave pubblica.
- C(T,A): dicitura usata per dire che si tratta del "certificato di A (Alice), rilasciato da T (la CA)"

Bisogna dire che la certificazione si basa sul meccanismo definito "Trusted Delegation", ossia:

- Ogni soggetto che usa un certificato, delega alla CA la fiducia di verificare l'identità di un altro soggetto e di attestare l'autenticità della rispettiva chiave.
- Un soggetto ha fiducia della chiave pubblica della CA. Se infatti mi fido della chiave pubblica della CA, per transitività mi fido anche dei certificati da essa rilasciati.

Per potersi fidare di un certificato, basandosi sulla Trusted Delegation, un soggetto (Bob) all'atto della ricezione di un certificato da parte di un altro soggetto (Alice) che vuole comunicare con esso, dovrà effettuare un *processo di verifica*:

1. Bob si procura la chiave pubblica della Certification Authority (*one time*);

2. Bob si procura l'identificatore unico di Alice;

3. Bob si procura il certificato di Alice, C(T,A)

4. Bob verifica il certificato

   (a) Bob verifica la validità della chiave della CA

   (b) Bob verifica che il certificato C(T,A) sia ancora valido

   (c) Bob verifica la firma su C(T,A) usando la chiave pubblica della CA

   (d) Bob verifica che il certificato C(T,A) non sia stato revocato

5. Se tutte le verifiche hanno esito positivo, allora Bob può accettare come autentica la chiave pubblica di Alice

Infine, analizziamo l'ultimo aspetto di nostro interesse legato al mondo delle infrastrutture a chiave pubblica, lo *stato di un certificato* e poi vediamo più in dettaglio un esempio di certificato che andremo ad usare successivamente nel corso della soluzione tecnica adottata.

Stato dei certificati Un certificato può trovarsi in due stati:

1. Scaduto (*expired*): quando termina il suo periodo di validità (L); questo significa che la chiave è ancora sicura dal punto di vista dell'algoritmo ma per qualche altra ragione risulta insicuro usarla. Un certificato in questo stato perde ogni valore legale.

2. Revoked: quando la chiave viene compromessa (viene rivelata o il soggetto cambia azienda) prima della scadenza del periodo di validità, il certificato viene marcato con questo stato e aggiunto a delle liste pubbliche dette CRL (Certificate revocation list). Tali liste sono create e gestite dalla CA (vi è apposta la firma della CA) e vengono rese note periodicamente permettendo la verifica offline dei certificati. L'unico svantaggio rilevabile è che, un avversario utilizza una chiave pubblica revocata fino alla prossima distribuzione della CRL.

Certificati X509.v3 Lo standard X509[19] fornito dall'International Telecomunications Union (ITU-T) definisce i formati standard per i certificati a chiave pubblica. Attualmente si è giunti alla versione 3 che definisce la struttura di un certificato digitale come un insieme di attributi ed estensioni:

- Certificato
  - Versione: numero di versione del certificato, attualmente la 3.
  - Numero seriale: numero seriale che identifica il certificato in modo univoco all'interno della CA
  - ID algoritmo: identificativo dell'algoritmo usato dalla CA per firmare il certificato
  - Issuer: identifica l'entità che ha firmato ed emesso il certificato. Contiene un Distinguished Name (DN) e un Issuer Distinguished Name.
  - Validità : l'ora e le date di inizio del periodo di validità del certificato

        * Non prima
        * Non dopo
    - Soggetto: nome
    - Informazioni sulla chiave pubblica del soggetto
        * Algoritmo per l'utilizzo della chiave pubblica
        * Chiave pubblica
    - Estensioni (facoltativo)

- Algoritmo di firma del certificato
- Firma del certificato

Le estensioni più comuni che si trovano nell'ambito dei certificati in formato X509 sono:

- .CER: certificato codificato con DER, a volte sequenze di certificati
- .DER: certificato codificato con DER
- .PEM: certificato codificato con Base64 (è un sistema di conversione da formato ASCII a binario che usa 64 simboli), racchiuso tra "BEGIN CERTIFICATE" e "END CERTIFICATE".
- .P12: PKCS #12 (può contenere certificati e chiavi pubbliche e private, ed è infatti usato per scambiarsi oggetti pubblici e privati, come le chiavi)
- .JKS: repository per certificati di sicurezza usato in Java.

3.5.2 Analisi di differenti soluzioni per la gestione dei certificati

Sulla base delle richieste progettuali, si è reso dunque necessario trovare delle soluzioni adatte per utilizzare i certificati all'interno delle componenti del sistema. Gli approcci seguiti sono stati principalmente due:

3.5.2.1 Generazione di certificati self-signed

Per questa tipologia di generazione è stato usato il tool "keytook" fornito da Java. Sono stati creati certificati self-signed (auto firmati), ossia l'equivalente di un certificato appartenente ad una CA root. In questo primo approccio l'idea era capire come instaurare la comunicazione, dunque è stata usata la seguente procedura per generare due coppie di chiavi (privata e pubblica) all'interno di file di store JKS; da ognuno dei due store è stato esportato il certificato e importato nell'altro così da garantire il trust nei processi di verifica. In particolare la procedura è caratterizzata dai seguenti step:

1. Su Windows, da prompt di comando digitare "keytool -genkey -keyalg RSA -alias connector -keystore connectorkeystore.jks" per genarare la coppia all'interno dello store e relativo certificato. A questo punto verrà richiesto di inserire tutte le info relative al certificato (vedi Figura 3.13). Questo va fatto anche per lo store dell'altro componente con cui si vuole comunicare, nel nostro caso la piattaforma.

2. Esportare il certificato dallo store del connector ed importarlo come "trusted certificate" in quello della piattaforma. Lo stesso si fa con la

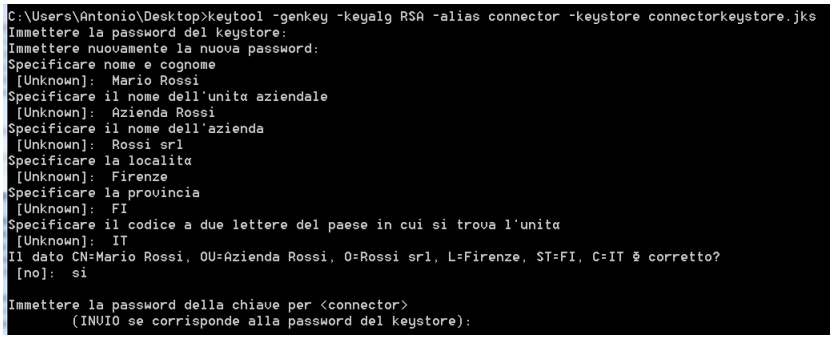


Figura 3.13: keytool - Generazione certificato self-signed

piattaforma nei confronti del connector. Usare i seguenti comandi:

- "keytool -export -alias connector -keystore connectorkeystore.jks -file connectorcert.cer" per esportare il certificato in un file .cer
- "keytool -import -alias connector -trustcacerts -keystore platformkeystore.jks -file connectorcert.cer" per importare il file .cer come certificato fidato

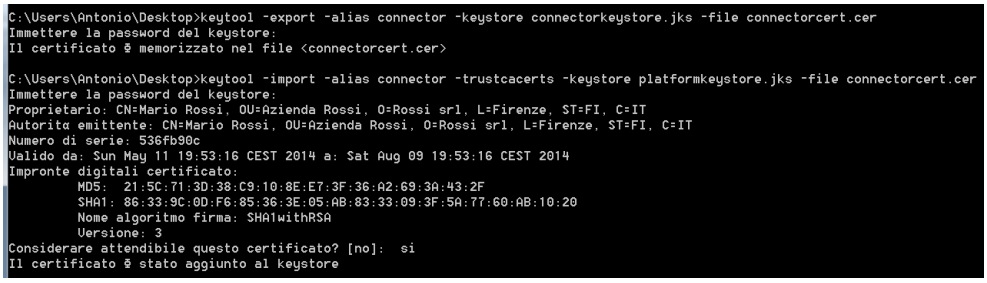


Figura 3.14: keytool - Procedura di export e import

Una volta pronti gli store, è infine stato possibile utilizzarli andando a referenziarli all'interno delle rispettive applicazioni, specificando la posizione

dello store sul filesystem e la password per la lettura dello stesso.

3.5.2.2 EJBCA

Si tratta di una Certification Authority basata su tecnologia Java che sfrutta le funzionalità di un Application Server quale JBoss. Realizza una PKI completa con funzioni di emissione e revoca dei certificati. Alcune delle caratteristiche principali sono:

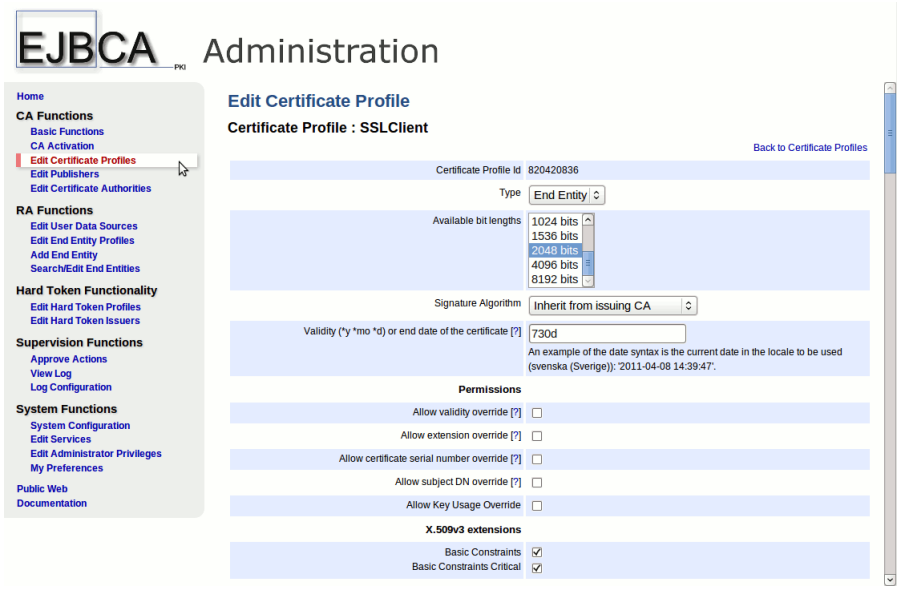


Figura 3.15: Immagine d'esempio di EJBCA - Pannello d'amministrazione

- Architettura flessibile
- Possibilità di avere più CA all'interno di una singola istanza di Ejbca.
- Supporto per RSA fino a 4096 bit

- Supporto per algoritmi hash come SHA-1, SHA-256 e MD5.
- I certificati Server e Client possono essere esportati in vari formati tra i quali PKCS12, JKS o PEM
- Gestione delle funzionalità di CA da interfaccia grafica (tramite browser)
- Supporto a token e smart card
- Possibilità di gestire livelli diversi di amministrazione corrispondenti a privilegi differenti
- Supporto allo standard X.509
- Supporto alla tecnologia OCSP (Online Certificate Status Protocol)
- Gestione e creazione di CRL (Certificate Revocation List)
- Supporto a vari tipi di database come MySQL, PostgreSQL

Inoltre è una PKI Open Source e su richiesta del team di progetto è stato ricercato software libero; EJBCA ha infatti licenza LGPL.

Capitolo 4

Scelte implementative e motivazioni

Sulla base delle opzioni presentate nel capitolo precedente si è optato per la soluzione che meglio si adattava alle richieste progettuali; nel seguito quindi si spiegheranno le motivazioni che hanno portato alla scelta.

4.1 Pitagora: perché SSL è preferibile a WS-Security

Quando si usa un'architettura a servizi realizzati per mezzo di Web Services, come abbiamo precedentemente visto, tali servizi sono spesso resi disponibili per mezzo di messaggi SOAP trasportati su protocollo HTTP. L'obiettivo è stato da subito la sicurezza di tali messaggi e quindi quale dei due approcci sia più idoneo alla specifica architettura del progetto PITAGORA è stato dettato dalle seguenti ragioni:

1. **Overhead in termini di messaggio**: dato che WS-Security opera una sicurezza a livello di messaggio è necessario l'uso di tag aggiuntivi all'interno del messaggio SOAP, nella fattispecie i tag <security> visti precedentemente. Questo comporta tempi maggiori a causa del processo di parsing dei messaggi[20]. Essendo richiesta una buona performance relativamente ai tempi di risposta del sistema è preferibile evitare questo overhead.

2. **Grado di conoscenza della tecnologia e facilità d'uso da parte dell'organizzazione**: sia SSL che WS-Security sono supportati da librerie che ne permettono l'uso all'interno dei Web Service ma quest'ultimo risulta più complesso da integrare, poiché richiede tipicamente l'uso di classi filtro (o interceptor) da implementare per la corretta manipolazione dei messaggi. SSL invece richiede più un approccio configurativo e meno implementativo.

3. **Comunicazione point-to-point vs end-to-end**: non essendo necessario l'attraversamento di nodi intermedi di elaborazione dei messaggi, in riferimento all'architettura di Figura 3.1 è sufficiente una comunicazione point-to-point, quindi SSL è idoneo per il nostro scopo.

4. **Operazioni crittografiche sul messaggio**: quando un applicazione mantiene una connessione aperta per effettuare molteplici richieste verso un Web Service, SSL fornisce migliori performance rispetto a WS-Security. La differenza computazionale è dovuta al fatto che: SSL, per le connessioni, usa principalmente crittografia a chiave simmetrica una volta che ha instaurato la sessione tramite l'handshake; WS-Security in-

vece necessita l'uso di crittografia asimmetrica su ogni messaggio (una computazione per la cifratura e una computazione per il processo di firma). E' noto che la crittografia simmetrica sia computazionalmente più rapida di quella asimmetrica e questo garantisce delle buone performance per il nostro sistema.

5. **Praticità d'uso**: sottoposto all'attenzione del team di progetto, la soluzione preferita è quella basata su SSL per ragioni di effort, praticità d'uso e conoscenza pregressa della tecnologia.

Sulla base di queste motivazioni, è stato dunque scelto l'approccio orientato ad una sicurezza a livello di trasporto, introducendo laddove necessario (lato servizio e lato connettori) le idonee misure e configurazioni perché ciò avvenisse. E' stato cioè scelto l'uso del protocollo **SSL con mutua autenticazione**. Al termine della messa in opera della soluzione è stata effettuata un'analisi di performance, al variare dei linguaggi e dell'uso della sicurezza o meno, al fine di capire come l'introduzione di tale sicurezza andasse ad intaccare le performance della comunicazione.

4.2 Pitagora: utilizzo di una CA interna

Dover generare e gestire certificati è un compito delicato ma necessario, dovendo instaurare delle comunicazioni sicure basate su SSL, e ulterioriormente dovendo fornire certificati ai partner per accedere alla Piattaforma. L'idea è fondamentalmente poter avere il controllo sul rilascio dei certificati in modo da permettere o inibire l'accesso alla piattaforma. Tale controllo è affidato all'owner della Piattaforma, ossia THALES. In sintesi, solo chi è in possesso

di un certificato rilasciato dall'owner può accedere alle funzionalità fornite dall'interfaccia di PITAGORA:

L'uso di una Certification Authority interna ha il vantaggio di fornire il completo controllo sul dominio di sicurezza associato alla Piattaforma, permettendo di:

- *gestire il rilascio*: un nuovo partner che richiede l'integrazione del suo modulo, ovvero vuole comunicare con la piattaforma, si rivolgerà all'owner della stessa, il quale previa identificazione, gli rilascerà le credenziali corrette. Nella fattispecie verrà rilasciato al partner la coppia di chiavi di cui la pubblica certificata e il certificato della CA per garantire i trust nella fase di verifica dell'handshake.

- *gestire le revoche*: ha una duplice funzione: quella di sicurezza, qualora un partner si accorgesse che le sue chiavi sono state compromesse può fare richiesta di revocarle e farsene generare una coppia nuova; quella di inibizione dell'accesso, ossia l'owner (quindi THALES) può agire sulla CA e bloccare le comunicazioni di un partner.

Essendo richiesta una gestione a 360° di una CA e allo stesso tempo che sia user-friendly, possibilmente utilizzando un'interfaccia grafica, delle due soluzione viste nel precedente capitolo, la seconda è sicuramente più idonea e quindi è stata di fatto scelta.

Capitolo 5

Soluzione proposta

La soluzione qui esposta prevede:

- la messa in opera della Certification Authority selezionata nel precedente capitolo, ossia EJBCA[1];
- la realizzazione di un prototipo di servizio, implementato con tecnologia Java e deployato su ServiceMix all'interno del container OSGi, e le relative configurazioni per abilitare la sicurezza tramite protocollo SSL/TLS;
- la realizzazione di due connettori, con tecnologie Java e C++ e le relative configurazioni di sicurezza;
- la preparazione di un piano di testing dei requisiti di sicurezza;

[1]Reperibile al seguente link: `http://www.ejbca.org`

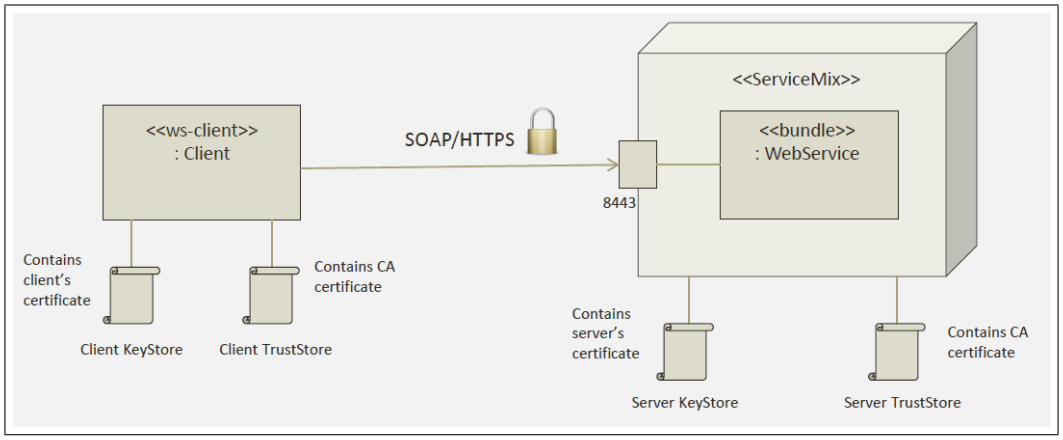


Figura 5.1: Design ad alto livello del prototipo

Una volta ottenuto il prototipo e testata la sua efficacia è stato possibile applicare le medesime accortezze sulla Piattaforma attualmente in uso, che sarà presentata come dimostratore presso la Regione.

5.1 Soluzione adottata per la CA

5.1.1 Installazione

L'installazione della release software EJBCA è stata testata su macchine con sistema operativo Unix/Linux, per la precisione sia su CentOS 6.5 x64 che Ubuntu 12.04 server x64. Una volta installato il sistema operativo su macchina fisica o virtuale (ad esempio usando VMware), i successivi componenti necessari per finalizzare l'installazione della CA sono:

- Software della CA: in questo caso EJBCA Community Edition 6.0.3;

- Application server nel quale avverrà il deploy della CA: in questo caso JBoss 7.1.1;
- OpenJDK 7 e Ant: qualora non fossero già presenti di default nella release del sistema operativo.

Si può passare all'installazione (si farà riferimento ad Ubuntu) secondo la seguente procedura:

1. Una volta volta installato il sistema operativo, aprire un nuovo terminale (chiamiamolo " ejbca"), accedere come root e portarsi nella propria home directory;
2. Installare il software necessario tramite repository Ubuntu:
   - sudo apt-get install openjdk-7-jdk ant ant-optional unzip ntp
3. Installare il software necessario non presente nei repository Ubuntu
   - wget `http://download.jboss.org/jbossas/7.1/jboss-as-7.1.1.Final/jboss-as-7.1.1.Final.zip`
   - wget `http://softlayer-ams.dl.sourceforge.net/project/ejbca/ejbca6/ejbca_6_0_3/ejbca_ce_6_0_3.zip`
   - unzip jboss-as-7.1.1.Final.zip
   - unzip ejbca_ce_6_0_3.zip
4. Configurare EJBCA così che possa trovare l'application server (JBoss)
   - echo "appserver.home=/root/jboss-as-7.1.1.Final">> ejbca_ce_6_0_3/conf/ejbca.properties

5. Aprire ora un secondo terminale (chiamiamolo "jboss") e avviare JBoss
    - jboss-as-7.1.1.Final/bin/standalone.sh
6. Da terminale "ejbca", effettuare il build e il deploy su JBoss
    - cd ejbca_ce_6_0_3
    - ant deploy
    - attendere che JBoss si riavvii
7. Sempre da terminale "ejbca" effettuare l'install per creare una CA iniziale e il keystore TLS per accedere ad essa
    - ant install
    - A questo punto compilare i campi che si presentano secondo le proprie preferenze o eventualmente scegliere tutti i valori di default
8. Ritornare sul terminale "jboss" e riavviare JBoss
    - ctrl-c
    - jboss-as-7.1.1.Final/bin/standalone.sh
9. Una volta terminato il setup della CA, copiare il file /root/ejbca_ce_6_0_3/p12/superadmin.p12 sulla macchina desktop dell'amministratore e importarlo nel browser web come certificato valido. Questo file fornisce le credenziali per accedere al pannello di amministrazione di EJBCA.

10. Dalla macchina desktop dell'amministratore, dal browser, accedere all'indirizzo https://ipserver:8443/ejbca, dove "ipserver" è l'ip della macchina su cui è stata installata la CA (vedi Figura 5.2).

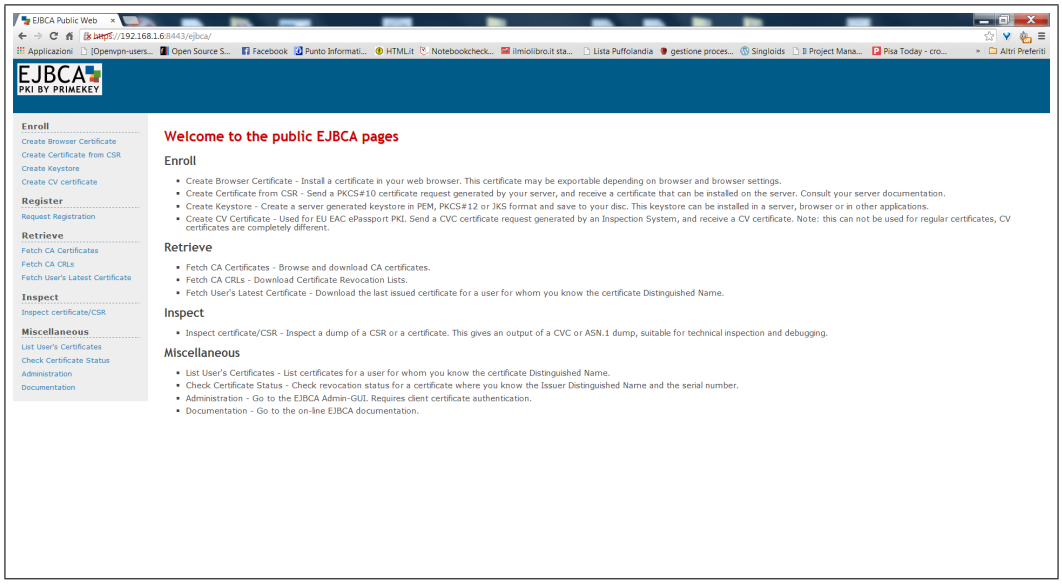


Figura 5.2: Homepage di EJBCA una volta installato

5.1.2 Configurazione

A questo punto accedendo al pannello di amministrazione è possibile usufruire di tutte di funzionalità legate alla gestione della CA; in particolare sarà già attiva una CA root, configurata secondo i parametri inseriti in fase di installazione. A questo punto, con la CA attiva, bisogna configurare due aspetti chiave per la successiva generazione dei certificati: il **profilo dei certificati**

Figura 5.3: EJBCA: Funzionalità di amministrazione

che si andranno a generare e il **profilo degli utenti** associati alla CA (vedi Figura 5.3).

Per la creazione del *profilo certificati*:

- Da pannello d'amministrazione, Certificate Profiles, inserire un nome nella casella (ad es. SSL2WAY) e quindi Add. In alternativa è possibile basarsi anche su template pronti, così come abbiamo fatto basandoci sul profilo Enduser.

- A questo punto è possibile impostare ogni nostra preferenza sul certificato che si andrà a rilasciare. Per il nostro caso sono di interesse solo alcune voci, le restanti possono essere come nel template:

  – Type: End Entity

  - Available bit lengths: da 1024 in su
  - Signature Algorithm: Inherit from issuing CA
  - Validity or end date of the certificate: 730d
  - Key Usage: Digital Signature, Non-repudiation, Key encipherment, Data encipherment
  - Extended Key Usage: Server Authentication, Client Authentication
  - Available CAs: selezionare la nostra CA

- Premere Save e verificare che il profilo sia stato aggiunto alla Lista dei Profili di Certificato

Similmente per il *profilo utenti*:

- Da pannello di amministrazione, End Entity Profiles, inserire un nome nella casella (ad es. PitagoraUser) e quindi Add.
- E' possibile adesso impostare ogni nostra preferenza sul profilo. Lasciando le impostazioni predefinite, per il nostro caso andremo semplicemente a legare il profilo utente con il profilo certificato precedentemente creato, attraverso le seguenti voci:
  - Default Certificate Profile: SSL2WAY
  - Available Certificate Profiles: SSL2WAY
  - Default CA: la nostra CA
  - Available CAs: la nostra CA

  - Default Token: User Generated
  - Available Tokens: User Generated, P12 file, JKS file, PEM file

- Premere Save e verificare che il profilo sia stato aggiunto alla Lista dei Profili Utente

5.1.3 Fase di rilascio certificati

Quando un partner farà richiesta per un certificato, previa identificazione gli verrà generato e consegnato fisicamente su supporto digitale (es. pendrive). Ad esempio, nel caso di connettore Java, quello che si otterrà sarà uno store JKS contenente al suo interno: la chiave privata, la chiave pubblica certificata e il certificato della CA. La procedura, dopo aver identificato il soggetto, prevede l'inserimento dello stesso nella base di dati della CA, secondo la seguente modalità:

- Da pannello di amministrazione, Add End Entity e quindi poi compilare i campi associati all'utente, come ad esempio mostra la Figura 5.4.

- Una volta aggiunto, fare logout da amministratore e tornare alla pagina iniziale di EJBCA. Da qui selezionare Create Keystore e inserire utente e password fornite al passo precedente. Nella pagina successiva impostare la lunghezza della chiave (2048 bits) e il profilo del certificato (SSL2WAY) e infine cliccando su Enroll partirà il download dello store JKS. Una volta ottenuto il keystore, lo si potrà quindi consegnare su supporto digitale al partner che ne ha fatto richiesta.

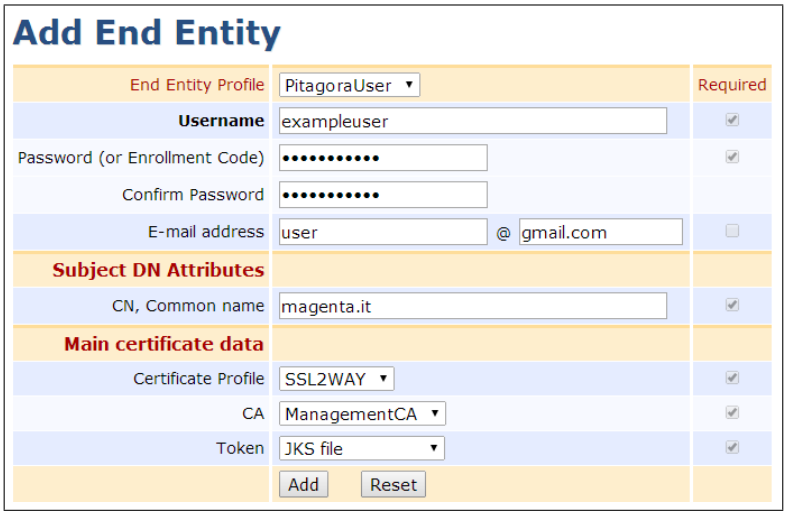


Figura 5.4: Add End Entity

Infine, come ultima nota, tramite la funzionalità di ricerca degli utenti all'interno del pannello di amministrazione, è possibile avere tutte le informazioni riguardo gli utenti e i certificati a loro rilasciati e sempre da qui possono essere effettuate le revoche di tali certificati. La Figura 5.5 mostra i campi di un certificato rilasciato dalla CA e come sia possibile effettuare la revoca in modo molto semplice.

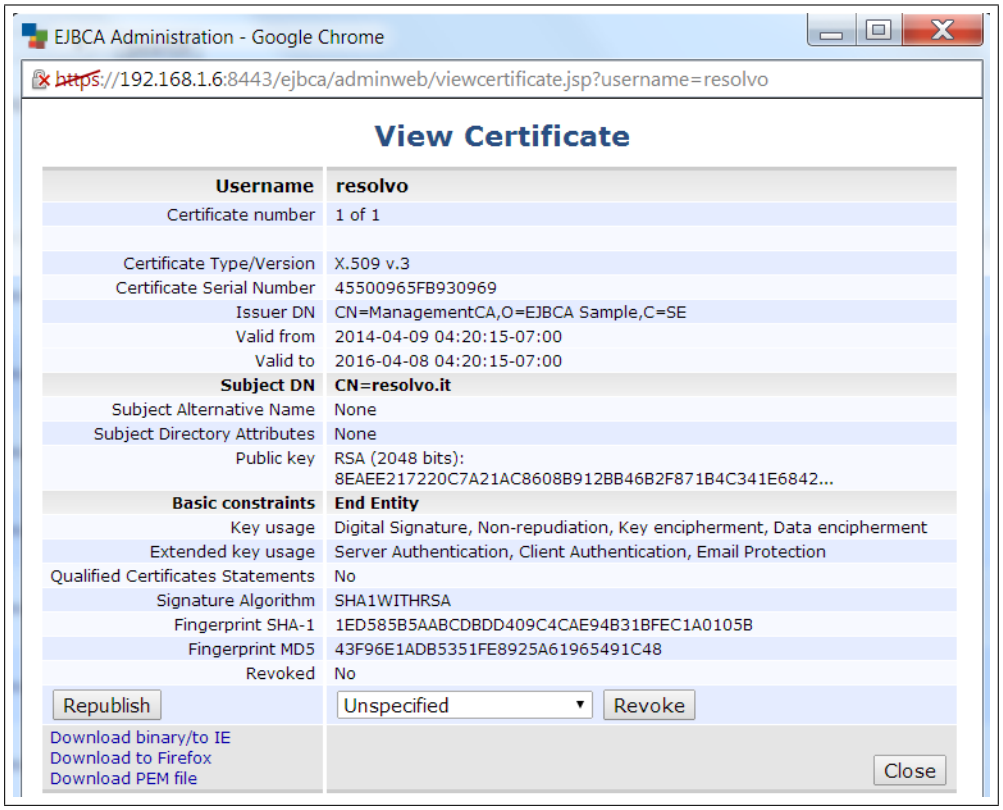


Figura 5.5: Visualizzazione di un certificato rilasciato

5.2 ServiceMix side

5.2.1 Creazione di Web Service su ServiceMix

Per creare il servizio web che useremo per testare la soluzione basata su SSL/TLS, ci si è avvalsi degli stessi strumenti e tecnologie alla base del progetto PITAGORA. In particolare per la realizzazione ci si è basati su tecnologie afferenti il mondo Java, quali:

- Apache CXF: un framework che ha l'obiettivo di fornire delle astrazioni e degli strumenti di sviluppo per esporre varie tipologie di servizi web. Realizza un'implementazione delle API JAX-WS.

- Spring: quale framework per lo sviluppo di applicazioni Java, viene utilizzato in combinazione ad Apache CXF e Apache CAMEL sfruttando il concetto di Dependency Injection, per mezzo di un approccio dichiarativo basato su file XML.

- Apache CAMEL: tramite l'uso di component e rotte, permette sia di esporre un servizio che di guidare la richieste verso tale servizio o verso altri endpoint.

Mentre per lo sviluppo, in linea con il team di progetto, sono stati usati i seguenti strumenti:

- Eclipse IDE, come ambiente di sviluppo

- Maven, per la gestione delle dipendenze e l'automatizzazione del processo di build

- SVN, per la condivisione con il team e la gestione delle versioni di progetto

5.2.2 Prototipo realizzato

Il servizio, come in precedenza esposto, è ottenuto come Web Service; realizza una logica semplice in quanto l'interesse era ottenere la sicurezza sul canale, qualsiasi fosse il dato in transito. Il servizio permette di memorizzare e

recuperare da una propria struttura dati interna, oggetti di tipo **Data** definiti in maniera specifica nel nostro *data model*.

A queste funzionalità, è stato inserito un livello di sicurezza, operando su due fronti:

1. Configurazioni apportate a ServiceMix per l'uso di SSL/TLS.

2. Realizzazioni di classi di tipo CAMEL Processor per scopi di sicurezza, ad esempio identificare gli utenti che fanno richieste al servizio tramite il certificato associato alla relativa sessione SSL.

Il servizio, logicamente e praticamente (in termini di progetti Maven)prevede una separazione tra:

- data model: comprende interfaccia del servizio e gli oggetti POJO che il servizio supporta.

- implementazione: comprende le classi che forniscono le funzionalità esposte dall'interfaccia e dei file XML per una gestione del servizio di tipo dichiarativa.

In particolare l'interfaccia (vedi Figura 5.6), espone tre metodi (funzionalità):

- List<Data> getData(List<String> ids): in base alla lista di Id in ingresso, restituisce la lista dei rispetti oggetti Data.

- void setData(List<Data> data): memorizza gli oggetti Data in ingresso.

- List<String> listDataIds(): restituisce l'elenco degli Id associati agli oggetti Data memorizzati dal servizio.

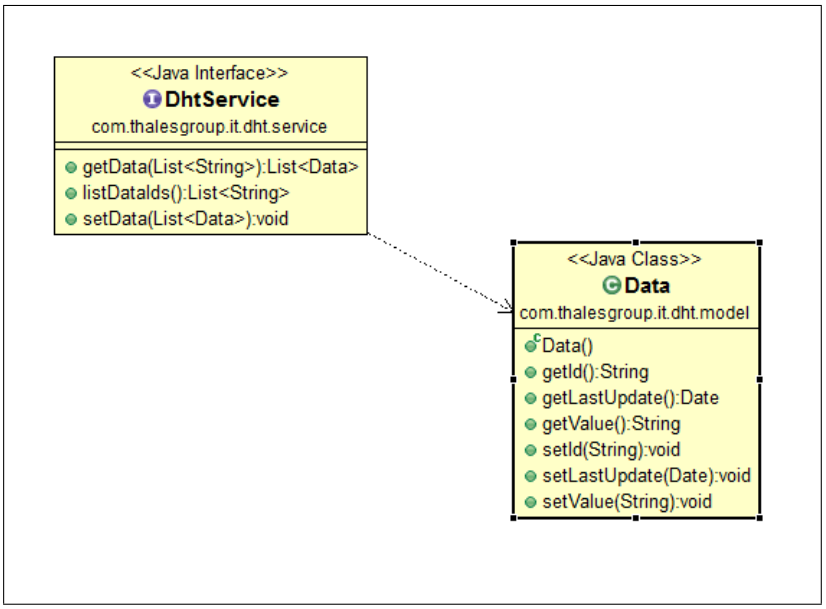


Figura 5.6: Diagramma delle classi del servizio

Una volta definito il data model, ci siamo concentrati sull'implementazione del servizio, in particolare:

- è stata definita una classe che implementi l'interfaccia e realizzi le semplici funzionalità previste.
- sono stati definiti due file XML.
- sono state realizzate le classi Processor legate a CAMEL utilizzate all'interno delle rotte definite nell'XML.

Questi file XML contengono informazioni utili al container OSGi per la gestione del servizio e per la sua esposizione. In particolare, per far si che il container, una volta che abbiamo installato il servizio al suo interno (ad esempio tramite hot deployment), lo renda disponibile ad una certa URI per gli utenti che ne vogliono usufruire, abbiamo usato un "component Camel CXF" che realizza un'istanza di Endpoint di servizio istruendo il container ad effettuare il binding del servizio all'URI: "*indirizzoServicemix*/cxf/dhtService" su tutte le porte disponibili.

```
<cxf:cxfEndpoint id="dhtServiceEndpoint"
    serviceClass="com.thalesgroup.it.dht.service.DhtService"
   address="/dhtService">
</cxf:cxfEndpoint>
```

Listing 5.1: Uso di CamelCXF Component per esporre un servizio web

Mentre il primo file XML ci è utile per l'esposizione del servizio, l'altro ha la funzione di definire i percorsi che i messaggi in arrivo all'Endpoint dovranno effettuare. Il file è stato così definito:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
  xmlns:cxf="http://camel.apache.org/schema/cxf" xmlns:util="
   http://www.springframework.org/schema/util"
  xmlns:camel="http://camel.apache.org/schema/spring"
   xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
   -2.5.xsd
        http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-spring.xsd
        http://camel.apache.org/schema/cxf
        http://camel.apache.org/schema/cxf/camel-cxf.xsd
        http://www.springframework.org/schema/util
```

```
16          http://www.springframework.org/schema/util/spring-util
    -2.5.xsd
17          http://www.springframework.org/schema/context
18          http://www.springframework.org/schema/context/spring-
    context-2.5.xsd ">
19
20  <!-- CAMEL Processors -->
21  <bean id="IdentityConnectorRetrieverProcessor"
22     class="com.thalesgroup.it.dht.processor.
    IdentityConnectorRetrieverProcessor" />
23  <bean id="responceProcessor" class="com.thalesgroup.it.dht.
    processor.ResponceProcessor" />
24  <bean id="voidResponceProcessor" class="com.thalesgroup.it.dht
    .processor.VoidResponceProcessor" />
25  <bean id="revocationControlWithCRL" class="com.thalesgroup.
    processor.RevocationControlWithCRL" />
26
27  <!-- Web Service implementation -->
28  <bean id="dhtMemoryService" class="com.thalesgroup.it.dht.
    MemoryService" />
29
30  <!-- Custom Exception -->
31  <bean id="notAuthException" class="com.thalesgroup.it.dht.
    service.fault.OperationFault">
32     <constructor-arg index="0" type="java.lang.String"
33        value="AuthorizationException" />
34  </bean>
35
36  <bean id="crlManager" class="com.thalesgroup.it.
    CRLRevocationManager">
37     <constructor-arg index="0" value="http://192.168.1.6:8080/
    ejbca/publicweb/webdist/certdist?cmd=crl&amp;issuer=CN%3
    dManagementCA%2cO%3dEJBCA+Sample%2cC%3dSE"/>
38  </bean>
39
40
41  <camelContext id="camelContext" xmlns="http://camel.apache.org
    /schema/spring">
42
43     <!--  Timer to get the CRL every 300 seconds -->
44     <route>
45        <from uri="timer://foo?fixedRate=true&amp;period=300000" /
    >
46        <to uri="bean:crlManager?method=storeCRL" />
47     </route>
```

```
48
49    <route>
50      <from uri="cxf:bean:dhtServiceEndpoint" />
51
52      <process ref="IdentityConnectorRetrieverProcessor" />
53
54      <process ref="revocationControlWithCRL" />
55
56      <choice>
57        <when>
58          <simple>${in.header.CRLMethodResult} == 'yes'</simple>
59          <throwException ref="notAuthException" />
60        </when>
61      </choice>
62
63      <!-- Saving operationName in CamelBeanMethodName header,
    CAMEL is able -->
64      <!--  to understand which is the right method to invoke.
    If the message contains -->
65      <!--  the header CamelBeanMethodName then that method is
    invoked, converting the -->
66      <!--  body to the type of the methods argument -->
67
68      <setHeader headerName="CamelBeanMethodName">
69        <simple>${in.header.operationName}</simple>
70      </setHeader>
71
72      <!-- Using dataformat=POJO the body of the IN message is
    represented by -->
73      <!--  a MessageContentList object. Each element
    MessageContentList is a list itself. -->
74      <!--  Data objects sent by connector to the service are
    stored in the first element -->
75      <!--  of MessageContentList -->
76
77      <convertBodyTo type="java.lang.Object[]" />
78
79      <choice>
80        <when>
81          <simple>${in.header.CamelBeanMethodName} == 'getData'<
    /simple>
82          <to uri="bean:dhtMemoryService?multiParameterArray=
    true" />
83        </when>
84        <when>
```

```
85          <simple>${in.header.CamelBeanMethodName} == '
    listDataIds'</simple>
86          <to uri="bean:dhtMemoryService?multiParameterArray=
    true" />
87        </when>
88        <when>
89          <!-- When the requestor is a connector, store data
    into this SMX and
90            propagate them to all the others SMX connected. -->
91          <simple>${in.header.CamelBeanMethodName} == 'setData'
    &amp;&amp;
92            ${in.header.EntityTypeConnected} == 'connector'
93          </simple>
94          <to uri="seda:split?waitForTaskToComplete=Never&amp;
    multipleConsumers=true" />
95        </when>
96        <when>
97          <!-- When the requestor is a SMX, just store data. -->
98          <simple>${in.header.CamelBeanMethodName} == 'setData'
    &amp;&amp;
99            ${in.header.EntityTypeConnected} == 'esb'
100         </simple>
101         <to uri="bean:dhtMemoryService?multiParameterArray=
    true" />
102       </when>
103     </choice>
104
105     <!-- Finally, to avoid any fault, service response is
    repackaged into
106       a MessageContentList (empty for void methods or with
    data for not void methods)
107       and the latter inserted into the OUT message body. -->
108     <when>
109       <simple>${header.CamelBeanMethodName} == 'setData'</
    simple>
110       <process ref="voidResponceProcessor" />
111     </when>
112     <otherwise>
113       <process ref="responceProcessor" />
114     </otherwise>
115   </route>
116
117   <route>
118     <from
119       uri="seda:split?waitForTaskToComplete=Never&amp;
```

```
    multipleConsumers=true" />
120       <to uri="bean:dhtMemoryService?multiParameterArray=true" /
    >
121     </route>
122
123     <route>
124       <from
125         uri="seda:split?waitForTaskToComplete=Never&amp;
    multipleConsumers=true" />
126       <to uri="bean:esb?method=setData&amp;multiParameterArray=
    true" />
127     </route>
128   </camelContext>
129 </beans>
```

Listing 5.2: File XML con rotte CAMEL definite per il servizio

Nella prima parte sono definiti i bean, cioè le classi che ci aspettiamo vengano istanziate da Spring quando il servizio viene avviato. Possiamo usare questi oggetti riferendoli con i loro id nella seconda parte, ossia all'interno del Camel Context, definendo le cosiddette rotte. Le rotte, come si può osservare, sono caratterizzate da un "from" e da uno o più "to", quindi hanno un punto di ingresso e più possibili destinazioni. Sono state definite quattro rotte ma solo la seconda definisce un punto di ingresso per i messaggi che giungono all'Endpoint, o più semplicemente al servizio. Le richieste SOAP quindi, una volta giunte presso l'Endpoint vengono deserializzate e trasformate in cosiddetti "CAMEL Message" così che possano fluire all'interno del contesto. Tali oggetti sono caratterizzati da una serie di attributi e un body.

In tutta questa elaborata definizione di rotte, il nostro ruolo in termini di sicurezza, è realizzato dall'implementazione di due Processor:

- **IdentityConnectorRetrieverProcessor**: considerando già configurato il protocollo SSL su ServiceMix (si vedrà nel prossimo paragrafo), questo Processor, estrae da un attributo del Camel Message l'informa-

zione per recuperare l'istanza di sessione SSL associata al messaggio. In questo modo, avendo configurato il protocollo per la mutua autenticazione, è possibile recuperare il certificato associato al richiedente e conoscerne l'identità. La conoscenza dell'identità può essere poi riusata successivamente per ulteriori elaborazioni o ad esempio per impedire a determinati utenti di eseguire determinate operazioni sul servizio. Nel nostro caso, una volta ottenuto il certificato del richiedente, lo memorizziamo come attributo del CAMEL Message per passarlo al Processor successivo che si occuperà del controllo della revoca.

```
1  public class IdentityConnectorRetrieverProcessor implements
       Processor {
2
3    @Override
4    public void process(Exchange exchange) {
5
6      /*
7       * Retrieve the SSL/TLS session extracting it from the
8       *  CAMEL_CXF_MESSAGE contained into the CAMEL Exchange
9       * Note: the CAMEL_CXF_MESSAGE contains informations
       about
10      * the request received on the cxf endpoint
11      */
12     org.apache.cxf.message.Message cxfMessage =
13         exchange.getIn().getHeader(CxfConstants.
       CAMEL_CXF_MESSAGE,
14             org.apache.cxf.message.Message.class);
15     TLSSessionInfo tlsInfo =
16         (TLSSessionInfo) cxfMessage.get("org.apache.cxf.
       security.transport.TLSSessionInfo");
17
18
19     /* Extract the certificate chain associated with the SSL
       /TLS session */
20     Certificate[] certs = tlsInfo.getPeerCertificates();
21
22     String certName = null;
23     String username = null;
24
```

```
25      if (certs == null || certs.length == 0) {
26        System.out.println("No client certificates were found");
27
28      } else {
29        X509Certificate[] x509Certs = (X509Certificate[]) certs;
30
31        /*
32         * Obtain the Distinguish Name and in particular the name of the module
33         */
34        certName = x509Certs[0].getSubjectX500Principal().getName();
35        username = this.splitDN(certName);
36
37        /*
38         * Set module identity and certificate retrieved from SSL session into
39         * CAMEL Exchange
40         */
41        exchange.getIn().setHeader("certificateToCheck", x509Certs[0]);
42        exchange.getIn().setHeader("EntityNameConnected", username);
43
44        if (username.matches("^thales.*$")) {
45          exchange.getIn().setHeader("EntityTypeConnected", "esb");
46        } else {
47          exchange.getIn().setHeader("EntityTypeConnected", "connector");
48        }
49      }
50    }
```

Listing 5.3: Processor CAMEL per il recupero delle identità dei connettori

- revocationControlWithCRL: Questo Processor, recupera il certificato memorizzato dall'IdentityConnectorRetrieverProcessor e ne verifica la presenza o meno all'interno della CRL presente all'interno del servizio. La CRL viene recuperata una volta all'avvio del servizio, tramite un

bean chiamato crlManager e ogni 300 secondi viene riscaricata, tramite richiesta alla Certification Authority, per mezzo di una rotta CAMEL temporizzata. In base allo stato del certificato viene settato un attributo all'interno del messaggio.

```
public class RevocationControlWithCRL implements Processor {
  public void process(Exchange exchange) throws Exception {

    /*
     * Get from context the bean that manage the CRL. Then verify the
     * revocation status.
     */
    CRLRevocationManager crm = exchange.getContext().getRegistry()
        .lookup("crlManager", CRLRevocationManager.class);

    if (crm.verifyCertificate((X509Certificate) exchange
        .getIn().getHeader("certificateToCheck"))) {
      exchange.getIn().setHeader("CRLMethodResult", "yes");
    } else
      exchange.getIn().setHeader("CRLMethodResult", "no");
  }
}
```

Listing 5.4: Processor CAMEL per la verifica della revoca

Se il certificato, associato all'utente che ha fatto la richiesta presso il servizio, risulta non revocato (si controlla l'attributo precedentemente settato) allora il messaggio prosegue fino a giungere all'implementazione del servizio. Al contrario se, se il certificato risulta revocato, viene rilanciata indietro all'utente un eccezione sotto forma di Fault SOAP.

```
<choice>
  <when>
    <simple>${in.header.CRLMethodResult} == 'yes'</simple>
    <throwException ref="notAuthException" />
  </when>
</choice>
```

Listing 5.5: Processor CAMEL per la verifica della revoca

5.2.3 Configurazione certficati SSL

Abbiamo finora parlato del servizio e dei suoi controlli iniziali come se il protocollo SSL fosse già abilitato su ServiceMix. Per farlo è necessario istruire l'HTTP Service interno a ServiceMix, fornito da PaxWeb, ad esporre il servizio su una determinata porta sicura. Per specificare questa direttiva basta configurare opportunamente un determinato file XML di configurazione. Tale file è localizzato nella directory "etc" dell'installazione di ServiceMix ed è chiamato jetty.xml. Si aggiunge un connettore sicuro alla configurazione come segue:

```
<Call name="addConnector">
    <Arg>
      <New class="org.eclipse.jetty.server.ssl.
  SslSelectChannelConnector">
          <Set name="keystore">C:/dev/EjbCA(certificati)/
  service/thales.jks</Set>
          <Set name="password">MD5:3
  a8aa14b09c007603f0c93151120b014</Set>
          <Set name="keyPassword">MD5:3
  a8aa14b09c007603f0c93151120b014</Set>
          <Set name="truststore">C:/dev/EjbCA(certificati)/
  service/thales.jks</Set>
          <Set name="trustPassword">MD5:3
  a8aa14b09c007603f0c93151120b014</Set>
      <Set name="protocol">TLSv1.2</Set>
      <Set name="needClientAuth">true</Set>
      <Set name="wantClientAuth">false</Set>
      <Set name="port">9443</Set>
      <Set name="maxIdleTime">300000</Set>
      <Set name="IncludeCipherSuites">
       <Array type="java.lang.String">
         <Item>TLS_DHE_RSA_WITH_AES_128_CBC_SHA</Item>
         <Item>SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA</Item>
         <Item>TLS_RSA_WITH_AES_128_CBC_SHA</Item>
         <Item>SSL_RSA_WITH_3DES_EDE_CBC_SHA</Item>
       </Array>
     </Set>
      </New>
   </Arg>
```

```
24      </Call>
```

Listing 5.6: Configurazione per abilitare SSL/TLS su ServiceMix su una determinata porta

In questo modo, quando andiamo ad installare il servizio su ServiceMix, esso verrà esposto sulla porta sicura 9443, per accettare richieste SSL in mutua autenticazione da validare utilizzando gli store JKS impostati che sono stati generati per mezzo della CA, come visto all'inizio del capitolo. La configurazione prevede la possibilità di restringere le CipherSuite, includendo ad esempio solo quelle accettate perché ritenute più sicure contro eventuali attacchi.

5.3 Partner Module side

Dopo aver configurato ServiceMix e installato il servizio web, in modo che possa gestire richieste basate su protocollo SOAP e trasportate su HTTPS, non resta che preparare i connettori. Il connettore, può essere pensato come uno strato di software che fa da tramite tra un modulo partner e la piattaforma; i suoi scopi infatti sono principalmente due:

1. Mappare le strutture dati specifiche del modulo di un partner nelle strutture dati della piattaforma.

2. Recuperare l'interfaccia del servizio e preparare la comunicazione sicura.

Sono stati realizzati per questo scopo connettori in diversi linguaggi, per dare prova della caratteristica di interoperabilità fornita dai Web Service e

quindi mostrare come la comunicazione possa avvenire anche tra componenti del sistema eterogenei. A tali connettori, sono state poi aggiunte le necessarie misure configurative per l'introduzione del protocollo SSL/TLS, e rilasciato un certificato valido ad ognuno di essi.

5.3.1 Comunicazione sicura JAVA based

Il connettore è stato sviluppato avvalendosi del framework Apache CXF, sia con approccio programmatico che dichiarativo tramite Spring. Per ognuno dei due casi si vedrà quindi quali sono state le necessarie configurazioni apportate.

5.3.1.1 Approccio programmatico

```
1
2  public class ClientDhtSecureCxf {
3    /**
4     * The main method.
5     *
6     * @param args the arguments
7     */
8    public static void main(String[] args) {
9
10     /*
11      * Set up a client proxy able to invoke the service.
       Endpoint address can be
12      * set dinamically.
13      */
14     JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
15     factory.create(DhtService.class);
16     factory.setAddress("https://192.168.3.196:9443/cxf/
       dhtService"); // ServiceEndpoint
17
18     DhtService client = (DhtService) factory.create();
19
20     /*
```

```
21      * Get the client interface and its conduit. Set the conduit
        to support
22      * SSL/TLS connection. Set the conduit with a
        MessageTrustDecider used to
23      * check revocation status of server's certificate.
24      */
25     final Client c = ClientProxy.getClient(client);
26     HTTPConduit http = (HTTPConduit) c.getConduit();
27     try {
28       SslSetupUtil.initializeConduitForSSL();
29       http.setTlsClientParameters(SslSetupUtil.getTlsParams());
30
31     } catch (Exception e) {
32       e.printStackTrace();
33     }
34
35     /*
36      * Invoke the service methods
37      */
38
39     Date date = new Date();
40     Random random = new Random();
41     List<Data> data = new ArrayList<Data>();
42
43     for (int i = 0; i < 100; i++) {
44       Data d = new Data();
45       d.setId(Integer.toString(i));
46       d.setValue("" + random.nextInt()); // random value
47       d.setLastUpdate(new Timestamp(date.getTime()));
48       data.add(d);
49     }
50
51     List<String> listID = new ArrayList<String>();
52     listID.add(new String("1"));
53     listID.add(new String("3"));
54     listID.add(new String("9"));
55     listID.add(new String("11"));
56     listID.add(new String("32"));
57     listID.add(new String("91"));
58
59     try {
60         /* setData */
61         client.setData(data);
62
63         /* getData */
```

```
64          List<Data> dataToGet = client.getData(listID);
65          for (Data dataResponce : dataToGet) {
66            System.out.println(dataResponce.getId() + " " +
    dataResponce.getValue());
67          }
68
69          /* listDataIds */
70          List<String> listing = client.listDataIds();
71          for (String listElement : listing) {
72            System.out.println(listElement);
73          }
74
75      } catch (SOAPFaultException e) {
76        e.printStackTrace();
77      } catch (Exception e) {
78        e.printStackTrace();
79      }
80    }
81  }
```

Listing 5.7: Esempio di connettore CXF

Il connettore prevede una dipendenza con il data model della piattaforma, in questo modo possiamo sfruttare un oggetto factory di CXF e istruirlo relativamente alla struttura dell'interfaccia e alla posizione del servizio (l'URI dell'Endpoint); quindi una volta fatto ciò la factory ci fornirà un modo semplice per fare chiamate al servizio. In questo modo abbiamo anche la libertà di poter impostare dinamicamente l'Endpoint qualora non sia noto a priori o dovesse cambiare nel tempo.

Successivamente viene recuperato l'oggetto HTTPConduit, specifico oggetto di CXF per la gestione dei protocolli HTTP e HTTPS, associato all'istanza di servizio recuperata prima. Di questo oggetto "conduit" vengono settati i parametri TLS all'interno di un oggetto TLSClientParameters, quali: versione del protocollo in uso, keystore e truststore contenenti certificati (definire entrambi abilita di fatto la mutua autenticazione) e chiavi da utiliz-

zare in fase di handshake, Ciphersuites accettate o meno. Qui di seguito una parte della classe che si occupa di inizializzare il conduit per la comunicazione sicura:

```
1
2 static final void initializeConduitForSSL() throws Exception {
3
4     try {
5
6       /* Set up the Keystore and Password */
7       KeyStore keyStore = KeyStore.getInstance("JKS");
8       String trustpass = "resolvo";
9       /* Provide your truststore */
10      File truststore = new File("C:/dev/EjbCA(certificati)/
    resolvo.jks");
11      keyStore.load(new FileInputStream(truststore), trustpass.
    toCharArray());
12
13      /* Set the SSL protocol */
14      tlsParams.setSecureSocketProtocol("TLSv1.2");
15
16      /*
17       * Set the trust store(decides whether credentials
    presented by a peer
18       * should be accepted)
19       */
20      TrustManagerFactory trustFactory = TrustManagerFactory.
    getInstance("SunX509");
21      trustFactory.init(keyStore);
22      TrustManager[] tm = trustFactory.getTrustManagers();
23
24      tlsParams.setTrustManagers(tm);
25
26      /* Set our Keystore (used if mutual authentication is
    required) */
27      KeyManagerFactory keyFactory = KeyManagerFactory.
    getInstance("SunX509");
28      keyFactory.init(keyStore, trustpass.toCharArray());
29      KeyManager[] km = keyFactory.getKeyManagers();
30      tlsParams.setKeyManagers(km);
31
32      /* Set all the needed include & exclude cipher filters */
33      FiltersType filter = new FiltersType();
34
```

```
35      filter.getInclude().add("TLS_DHE_RSA_WITH_AES_128_CBC_SHA"
    );
36      filter.getInclude().add("SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
    ");
37      filter.getInclude().add("TLS_RSA_WITH_AES_128_CBC_SHA");
38      filter.getInclude().add("SSL_RSA_WITH_3DES_EDE_CBC_SHA");
39
40      /*
41       * Export key exchange suites use authentication that can
    easily be
42       * broken.
43       */
44      filter.getExclude().add(".*_EXPORT_.*");
45
46      /*
47       * Suites with weak ciphers (typically of 40 and 56 bits)
    use encryption
48       * that can easily be broken.
49       */
50      filter.getExclude().add(".*_EXPORT1024_.*");
51      filter.getExclude().add(".*_WITH_DES_.*");
52      filter.getExclude().add(".*_WITH_RC4_.*");
53
54      /* NULL cipher suites provide no encryption. */
55      filter.getExclude().add(".*_WITH_NULL_.*");
56
57
58      /* Anonymous Diffie Hellman suites do not provide
    authentication */
59      filter.getExclude().add(".*_DH_anon_.*");
60
61      tlsParams.setCipherSuitesFilter(filter);
62
63    } catch (Exception e) {
64      throw new Exception("Failed to initialize TLSParameters
    for conduit!", e);
65    }
66  }
```

Listing 5.8: Parametri SSL/TLS per il Conduit

Infine vengono invocati i metodi definiti dall'interfaccia. E' possibile avere una visione diretta di tutto il processo di Handshake iniziale e cifratura, abilitando la funzione di debug della classe System a mostrare l'intero processo

SSL.

```
    /*
     * Show the SSL Handshake
     */
     System.setProperty("javax.net.debug", "ssl");
```

Listing 5.9: Abilitare il debug di SSL

5.3.1.2 Approccio dichiarativo

Per l'approccio dichiarativo, si usa la Dependency Injection, per cui sia il recupero del servizio che le sue configurazioni per supportare il protocollo SSL/TLS vengono esplicitate all'interno di un file XML.

```

<jaxws:client id="esb"
    serviceClass="com.thalesgroup.it.dht.service.DhtService"
    address="https://192.168.3.12:9443/cxf/dhtService" />

<http:conduit name="*.http-conduit">
    <http:tlsClientParameters disableCNCheck="false">
      <sec:keyManagers keyPassword="resolvo">
        <sec:keyStore password="resolvo" type="JKS"
          file="C:/dev/EjbCA(certificati)/resolvo.jks" />
      </sec:keyManagers>
      <sec:trustManagers>
        <sec:keyStore password="resolvo" type="JKS"
          file="C:/dev/EjbCA(certificati)/resolvo.jks" />
      </sec:trustManagers>

      <sec:cipherSuitesFilter>
        <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
-       <sec:exclude>.*_DH_anon_.*</sec:exclude>
        <sec:exclude>.*_EXPORT_.*</sec:exclude>
        <sec:exclude>.*_EXPORT1024_.*</sec:exclude>
        <sec:exclude>.*_WITH_DES_.</sec:exclude>
        <sec:exclude>.*_WITH_RC4_.</sec:exclude>
      </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
  </http:conduit>
```

Listing 5.10: Inizializzazione del connettere tramite Spring

Ora non resta che permettere a Spring di istanziare gli oggetti specificati nel file quando viene lanciata l'applicazione. Spring inizializza un suo contesto di programma all'avvio nel seguente modo:

```
public class Launcher {
  public static void main(String[] args) throws IOException {

    ApplicationContext context = new
    ClassPathXmlApplicationContext(
        "META-INF/spring/beans-routes.xml","META-INF/spring/
    beans-test.xml");

  /* Spring istanzia gli oggetti spcificati in beans-test.xml,
    quindi da qui in poi basta recuperarli
      dal contesto e usarli. */
}
```

Listing 5.11: Inizializzazione del connettere tramite Spring

5.3.2 Comunicazione sicura C/C++ based

Il connettore C++ è stato sviluppato tramite l'utilizzo di strumenti e tecnologie diverse rispetto agli approcci visti fin d'ora, in particolare si è fatto uso di:

1. Visual Studio 2013, come IDE di supporto allo sviluppo;
2. gSOAP 2.7.17, come toolkit per la traduzione del WSDL in classi C++ di supporto alla comunicazione;
3. OpenSSL 1.0.1g, quale libreria di sicurezza usata in combinazione con gSOAP;

5.3.2.1 Primo passo: usare gSOAP

Per prima cosa bisogna recuperare il WDSL associato al servizio. Quindi una volta avviato ServiceMix con il Web Service installato, possiamo recuperare il documento all'URI "*ipServiceMix/cxf/dhtService?wsdl*". Una volta in possesso del WSDL, tramite linea di comando, andiamo a costruire le nostre classi C++ sfruttando il toolkit gSOAP e in particolare i suoi due applicativi **wsdl2h** e **soapcpp2** nel seguente ordine e modo:

1. Compiliamo il WSDL:

```
wsdl2h -I \gsoap-2.7\gsoap\WS -f -o dhtService.h dhtService.
    wsdl
```

   wsdl2h.exe è l'applicazione parser di gSOAP. Il parametro *-I* indica al parser dove trovare gli "include files" di gSOAP necessari. Il parametro *-f* indica la volontà di avere codice C++ generato. Il parametro *-o* specifica il nome del file di output. Il percorso alla fine del comando indica il path del documento WSDL. Al termine, se l'esito è positivo, viene generato in output il file "dhtService.h".

2. Generiamo gli stub C++:

```
soapcpp2 -C -L -I \gsoap-2.7\gsoap\import -w -x dhtService.h
```

   soapcpp2.exe è l'applicazione stub generator di gSOAP. Il parametro *-C* dice al generatore di produrre solo codice client-side. Il *-L* di non produrre un file aggiuntivo di libreria non utile ai nostri scopi. Il parametro *-I* specifica il path dei file da includere necessari. I parametri finali invece indicano rispettivamente di non produrre uno "schema file" finale e non produrre degli XML message files. Al termine di questa

operazione verranno generati in uscita una serie di file che useremo all'interno del progetto del connettore.

5.3.2.2 Secondo passo: preparare il progetto C++

Da Visual Studio creare un progetto *C++ Win32 Console Application* mantenendo tutte le impostazioni di default. Aggiungere al progetto i file generati con il generator soapcpp2. Nello specifico:

- DhtServiceServiceSoapBinding.nmap
- soapC.cpp
- soapClient.cpp
- soapDhtServiceServiceSoapBindingProxy.h
- soapH.h
- soapStub.h

Aggiungere inoltre due file, prendendoli dalla directory di gSOAP:

- stdsoap2.cpp
- stdsoap2.h

Adesso possiamo includere la libreria OpenSSL al progetto in modo da usare le sue funzionalità nel codice. Le operazioni di inclusione sono:

- Sul progetto, tasto destro e accediamo alle sue Properties. Quindi andiamo su Configuration Properties, C/C++ and General e aggiungiamo i path che localizzano gSOAP e OpenSSL sul filesystem.

- Dalle Properties del progetto, Configuration Properties, C/C++ and Processor, aggiungiamo "WITH_OPENSSL" per abilitare SSL su gSOAP.
- Dalle Properties del progetto, Configuration Properties, Linker and General, aggiungiamo il path alle librerie di OpenSSL.
- Sempre dalle Properties del progetto, Configuration Properties, Linker and Input, aggiungiamo "libeay32.lib" e "ssleay32.lib" come Additional Dependencies.

5.3.2.3 Terzo passo: scrivere il codice

A questo punto il nostro progetto conosce l'interfaccia e i dati da scambiare, non resta che inizializzare a livello di codice il protocollo, aggiungendo al connettore la seguente porzione, in cui specificando la presenza di un certificato per il client, automaticamente abilitiamo il protocollo per la mutua autenticazione:

```
1  soap_ssl_init(); /* init OpenSSL  */
2    if (soap_ssl_client_context(proxy.soap,
3      SOAP_SSL_DEFAULT | SOAP_SSL_REQUIRE_SERVER_AUTHENTICATION,
4      "C:\\dev\\EjbCA(certificati)\\dhtuser.pem", /* keyfile:
       required only when client must authenticate to server */
5      NULL,  /* password to read the key file (not used with
       GNUTLS) */
6      "E:\\ManagementCA.cacert.pem",  /* cacert file to store
       trusted certificates (needed to verify server) */
7      NULL,  /* capath to directory with trusted certificates */
8      NULL    /* if randfile!=NULL: use a file with random data to
        seed randomness */
9      ))
10   {
```

Listing 5.12: Inizializzazione SSL per gSOAP

Una semplice versione del connettore che effettua una chiamata al Web Service installato in ServiceMix, inviandogli un dato di tipo Data, specifico del data model della piattaforma è qui mostrata. Unico dettaglio importante è impostare l'indirizzo dell'Endpoint, in quanto nel file WSDL potrebbe non essere presente.

```
#include "stdafx.h"
#include "stdsoap2.h"
#include "soapDhtServiceServiceSoapBindingProxy.h"
#include "DhtServiceServiceSoapBinding.nsmap"
#include <time.h>

const char defaultEndpoint[] = "https://192.168.1.4:9443/cxf/
    dhtService";

int _tmain(int argc, _TCHAR* argv[])
{

  DhtServiceServiceSoapBinding proxy;

  soap_ssl_init(); /* init OpenSSL (just once) */
  if (soap_ssl_client_context(proxy.soap,
    SOAP_SSL_DEFAULT | SOAP_SSL_SKIP_HOST_CHECK |
    SOAP_SSL_REQUIRE_SERVER_AUTHENTICATION,
    "C:\\dev\\EjbCA(certificati)\\client\\fromVirtualCA\\dhtuser
    .pem", /* keyfile: required only when client must
    authenticate to server*/
    NULL, /* password to read the key file (not used with
    GNUTLS) */
    "E:\\ManagementCA.cacert.pem", /* cacert file to store
    trusted certificates (needed to verify server) */
    NULL, /* capath to directory with trusted certificates */
    NULL /* if randfile!=NULL: use a file with random data to
    seed randomness */
    ))
  {
    soap_print_fault(proxy.soap, stderr);
    system("PAUSE");
    exit(1);
  }

  if (argc > 1) {
    proxy.endpoint = (const char*)argv[1];
```

```
31    }
32    else {
33      proxy.endpoint = defaultEndpoint;
34    }
35
36    std::cout << "Using Endpoint " << proxy.endpoint << std::endl;
37
38
39    // prepare a vector of Data as input of setData
40    ns1__Data d = ns1__Data();
41    d.id = "21";
42    d.value = "5542";
43    d.lastUpdate = time(0);
44
45    std::vector<ns1__Data*>datalist;
46    datalist.push_back(&d);
47
48
49    // prepare the input message
50    ns1__setData req;
51    req.datas = datalist;
52
53    // prepare the output message
54    ns1__setDataResponse resp;
55
56    try{
57
58      if (SOAP_OK == proxy.__ns1__setData(&req, &resp)) {
59        std::cout << "setData() DONE" << std::endl;
60      }
61      else {
62        soap_print_fault(proxy.soap, stderr);
63      };
64    }
65    catch (std::exception e){
66      std::cout << e.what() << std::endl;
67    }
68    system("PAUSE");
69    return 0;
70  }
```

Listing 5.13: Semplice Connettore C++

5.4 Prototipo su ServiceMix distribuiti

Task aggiuntivo rispetto all'idea di partenza è stato replicare il prototipo di servizio su più ServiceMix (SMX), installati su macchine fisiche distinte, e farli interagire tra loro sfruttando le configurazioni e le implementazioni precedentemente descritte. In questo semplice caso, oltre al ServiceMix iniziale ne è stato aggiunto un altro, ma ipoteticamente è possibile avere N ServiceMix collegati tra loro.

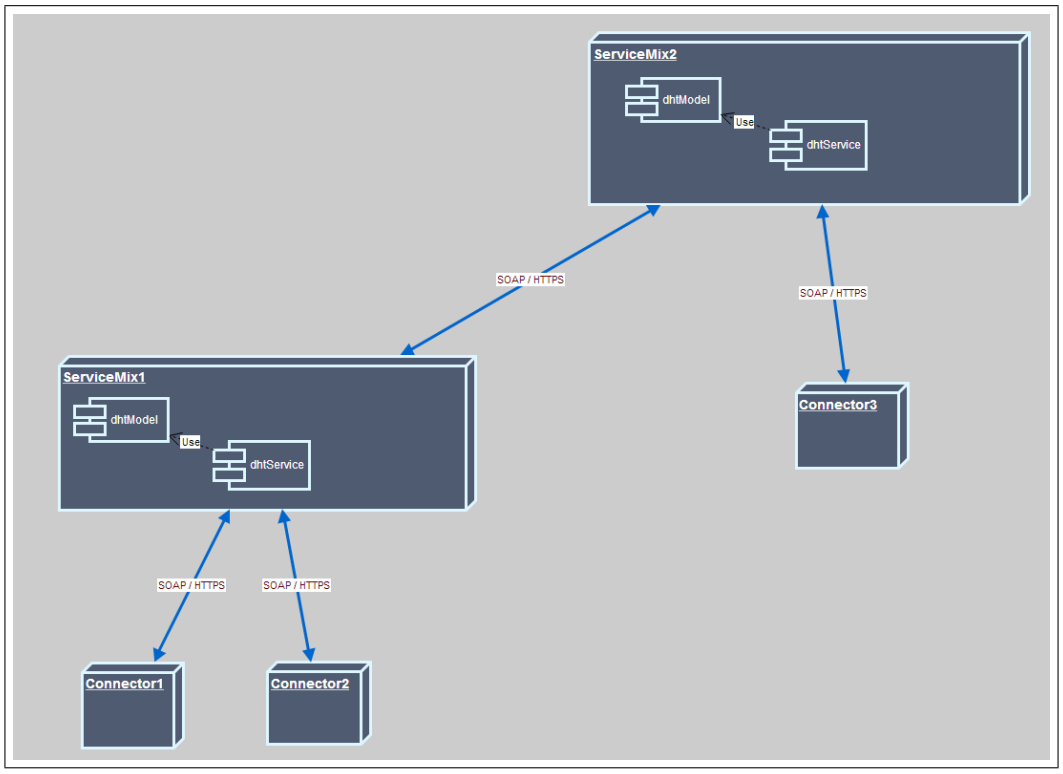


Figura 5.7: Rappresentazione ad alto livello di comunicazione distribuita tra più ServiceMix

E' stata così realizzata una HashMap distribuita, secondo la seguente lo-

gica: quando un connettore effettua una setData(), quindi vuole memorizzare un'informazione su un determinato ServiceMix (ad esempio ServiceMix1), il dato viene memorizzato sul ServiceMix in questione e viene inviato a tutti gli altri ServiceMix a cui è connesso. Si tratta quindi di avere in ogni momento lo stesso stato di memoria su ogni ServiceMix cosicché un connettore possa estrarre le informazioni anche da Endpoint differenti. Allo stesso modo con cui i connettori si collegano a ServiceMix, anche le comunicazioni inter-ServiceMix sono basate su protocollo SSL/TLS in mutua autenticazione.

Per realizzare quanto appena detto si è operato in tre punti all'interno del codice del servizio:

1. All'interno dell'IdentityConnectorRetrieverProcessor:

```
1       /*
2        * Set module identity and certificate retrieved from
    SSL session into
3        * CAMEL Exchange
4        */
5       exchange.getIn().setHeader("EntityNameConnected",
    username);
6
7       if (username.matches("^thales.*$")) {
8         exchange.getIn().setHeader("EntityTypeConnected", "
    esb");
9       } else {
10        exchange.getIn().setHeader("EntityTypeConnected", "
    connector");
11      }
```

Listing 5.14: Porzione di codice che identifica il tipo di entità che invoca il servizio

Una volta estratto il Distinguished Name dal certificato, da questo ricaviamo l'identificativo del richiedente e lo inseriamo in una variabile username. Definendo la semplice politica che, i certificati rilasciati ai

ServiceMix abbiano la forma thales1, thales2 ecc.., possiamo stabilire se il richiedente è un connettore o un esb e settare una variabile appropriata all'interno del CAMEL Message.

2. Nelle rotte CAMEL: dopo aver appurato l'identità e controllato che il certificato non sia revocato, la rotta conduce all'implementazione del servizio.

```
<choice>
        <when>
          <!-- When the requestor is a SMX, just store data.
     -->
          <simple>${in.header.CamelBeanMethodName} == '
    setData' &amp;&amp;
            ${in.header.EntityTypeConnected} == 'esb'
          </simple>
          <to uri="bean:dhtMemoryService?multiParameterArray
    =true" />
        </when>

        <when>
          <!-- When the requestor is a connector, store data
     into this SMX and
            propagate them to all the others SMX connected.
    -->
          <simple>${in.header.CamelBeanMethodName} == '
    setData' &amp;&amp;
            ${in.header.EntityTypeConnected} == 'connector'
          </simple>
          <to uri="seda:pluto?waitForTaskToComplete=Never&
    amp;multipleConsumers=true" />
        </when>
</choice>
```

Listing 5.15: Message Routing basato sul metodo invocato e sull'identità del richiedente

Se il richiedente è un ESB memorizzo il dato e concludo (primo blocco when), in modo da evitare loop tra gli ESB. Se il richiedente è un

connettore, splitto la richiesta (secondo blocco when) utilizzando una coda SEDA (component specifico di CAMEL) nel seguente modo:

```
<route>
  <from
    uri="seda:split?waitForTaskToComplete=Never&amp;
multipleConsumers=true" />
  <to uri="bean:dhtMemoryService?multiParameterArray=
true" />
</route>

<route>
  <from
    uri="seda:split?waitForTaskToComplete=Never&amp;
multipleConsumers=true" />
  <to uri="bean:esb?method=setData&amp;
multiParameterArray=true" />
</route>
```

Così facendo, la prima rotta memorizza il dato proveniente dalla coda nel ServiceMix mentre la seconda invoca un bean chiamato "esb" che farà da consumer nei confronti dell'altro ServiceMix presso cui voglio inviare l'informazione.

3. Definendo un CXF consumer: aggiungiamo un file XML al servizio, nel quale si definisce dichiarativamente, il bean esb che sarà il consumer (in sostanza un client) che chiamerà l'altro ServiceMix e gli passerà i dati.

```

<http:conduit name="*.http-conduit">
    <http:tlsClientParameters disableCNCheck="false">
      <sec:keyManagers keyPassword="thales">
        <sec:keyStore password="thales" type="JKS"
          file="C:/dev/EjbCA(certificati)/service/thales.jks
    " />
      </sec:keyManagers>
      <sec:trustManagers>
        <sec:keyStore password="thales" type="JKS"
```

```
10              file="C:/dev/EjbCA(certificati)/service/thales.jks
    " />
11          </sec:trustManagers>
12          <sec:cipherSuitesFilter>
13            <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
14 -          <sec:exclude>.*_DH_anon_.*</sec:exclude>
15            <sec:exclude>.*_EXPORT_.*</sec:exclude>
16            <sec:exclude>.*_EXPORT1024_.*</sec:exclude>
17            <sec:exclude>.*_WITH_DES_.</sec:exclude>
18            <sec:exclude>.*_WITH_RC4_.</sec:exclude>
19          </sec:cipherSuitesFilter>
20        </http:tlsClientParameters>
21      </http:conduit>
22
23      <jaxws:client id="esb"
24        serviceClass="com.thalesgroup.it.dht.service.DhtService"
          address="https://192.168.3.196:9443/cxf/dhtService" />
```

Listing 5.16: CXF consumer riferibile all'interno del servizio

Notare appunto l'id associato al client che coincide ovviamente con il nome del bean specificato nella seconda rotta. Questo bean prende come store di riferimento e quindi, come chiavi per effettuare la comunicazione, quelli del ServiceMix in cui si trova, in modo da farsi riconoscere come un richiedente ESB.

5.5 Best practices e Test dei requisiti

Per la configurazione del protocollo, sono state prese come riferimento una serie di best practices secondo le specifiche fornite da **OWASP (Open Web Application Security Project)** e relative ai Web Service. Tali specifiche forniscono linee guida per mettere in sicurezza un Web Service ed evitare possibili attacchi. In particolare:

- **Transport Confidentiality**: tutte le comunicazioni con e tra Web Services, contenenti dati sensibili, devono essere cifrate tramite protocollo TLS ben configurato. Questo perchè SSL/TLS fornisce numerosi benefici al di là della confidenzialità, quali controllo d'integrità, difesa da replay attacks e autenticazione delle parti. E' buona norma:
  - *Usare una Certification Authority appropriata*: bisogna evitare di presentare all'utente certificati firmati da un'autorità sconosciuta. Deve essere sempre possibile poter avere accesso al certificato della CA che ha effettuato il rilascio. Nel nostro caso forniamo noi il certificato della CA ed esso è pubblico e reperibile dai partner.
  - *Supportare solo "Strong Protocols"*: debolezze sono state identificate in SSLv2 e SSLv3. Buona norma per la sicurezza a livello di trasporto è utilizzare i protocolli TLS 1.0, TLS 1.1 o TLS 1.2. E' appunto utilizzato quest'ultima configurazione.
  - *Supportare solo "Strong Cryptographic Ciphers"*: Ogni protocollo fornisce delle Cipher Suite che vengono negoziate tra gli interlocutori e determinano la resistenza agli attacchi. E' stato configurato il prototipo in modo da forzare l'uso della cipher suite <RSA, AES128_CBC, SHA1> il cui uso è ritenuto sicuro.
  - *Disabilitare Rinegoziazione e Compressione*: al fine di impedire la possibile messa in atto di attacchi noti a TLS.
  - *Usare chiavi robuste*: generare chiavi di dimensione minima pari a 2048 bit. E' il valore utilizzato quando generiamo le chiavi con la nostra CA. E' possibile anche superare questo valore.

- **User Authentication**: verificare l'identità dell'utente che prova a connettersi al servizio. Si raccomanda l'uso di un'autenticazione basata su certificato del client in quanto ritenuta più robusta rispetto alla Basic Authentication.

- **Authorization**: un Web Service deve essere in grado di determinare se un client è autorizzato ad effettuare una determinata operazione su di esso. Successivamente all'autenticazione il Web Service deve controllare, per ogni richiesta, se il client è autorizzato. Questo avviene, nel nostro caso, in presenza di certificato revocato ma può essere esteso facilmente definendo delle policy nelle rotte CAMEL.

Infine, in quanto partner del progetto PITAGORA e sulla base del prototipo realizzato, le cui parti più rilevanti andranno ad inserirsi nella Piattaforma finale, sono stati preparati i seguenti test sui requisiti (autenticità, integrità e confidenzialità) per comprovare l'effettiva validità della soluzione proposta:

TEST ID: **PIT-STD-IVVP-001**

LOCATION OF THE TEST:	**Reference Platform**	ATTENDEES:	**Thales**
TEST TITLE		PURPOSE OF THE TEST	

REQUIREMENTS (Reference Id of the requirement(s) to be tested):

PIT-SSS-REQ-139

EXPECTED TYPE OF TEST:
Inspection

PURPOSE (Description of the function to be validated):

The platform must be able to identify trusted and non-trusted entities(modules) that want to communicate with it.

INITIAL CONDITION:

PITAGORA sw stack up & running

Certificates, released by the internal CA, installed on PITAGORA Platform and on partner module.

TEST INPUT (test trigger):

The module try to connect to the Platform

STEP	INPUT/ACTION	PASS CRITERIA	PASS	FAIL	COMMENTS
1.	A trusted module tries to connect to the Platform in order to use its services.	The SSL Handshake protocol is performed successfully and an SSL connection is established between the Platform and the module.			
2.	The module tries to perform an operation over the Platform.	The Platform returns to module a response consistent with the request.			

GLOBAL TEST RESULT

☐ PASS ☐ FAIL

COMMENTS: The SSL handshake protocol is analyzed by logs or by inspection of the channel.

Tester in charge:	DATE:

Figura 5.8: Test di autenticazione in presenza di certificato corretto

TEST ID: **PIT-STD-IVVP-001(B)**			
LOCATION OF THE TEST:	**Reference Platform**	ATTENDEES:	**Thales**
TEST TITLE		PURPOSE OF THE TEST	

REQUIREMENTS (Reference Id of the requirement(s) to be tested):

PIT-SSS-REQ-139

EXPECTED TYPE OF TEST:
Inspection

PURPOSE (Description of the function to be validated):

The platform must be able to identify trusted and non-trusted entities(modules) that want to communicate with it.

INITIAL CONDITION:

PITAGORA sw stack up & running

Certificates, NOT released by the internal CA, installed on partner module.

TEST INPUT (test trigger):

The module try to connect to the Platform

STEP	INPUT/ACTION	PASS CRITERIA	PASS	FAIL	COMMENTS
1.	An untrusted module tries to connect to the Platform in order to use its services.	The SSL Handshake protocol is performed unsuccessfully. A "not valid certificate error" is shown by the module, and no SSL connection is established between the module and the Platform.			

GLOBAL TEST RESULT

☐ PASS ☐ FAIL

COMMENTS: The SSL handshake protocol is analyzed by logs or by inspection of the channel.

Tester in charge:	DATE:

Figura 5.9: Test di autenticazione con certificato non corretto

TEST ID: **PIT-STD-IVVP-002**			
LOCATION OF THE TEST:	**Reference Platform**	ATTENDEES:	**Thales**
TEST TITLE		PURPOSE OF THE TEST	

REQUIREMENTS (Reference Id of the requirement(s) to be tested):

PIT-SSS-REQ-140
PIT-SSS-REQ-141

EXPECTED TYPE OF TEST:
Inspection

PURPOSE (Description of the function to be validated):

Data returned by the platform to applications must be accessible only to the legitimate users and not be modified in an unauthorized or undetected manner.

INITIAL CONDITION:
PITAGORA sw stack up & running
Handshake protocol completed and session established between module and PITAGORA Platform

TEST INPUT (test trigger):
The module send a data to the Platform.

STEP	INPUT/ACTION	PASS CRITERIA	PASS	FAIL	COMMENTS
1.	A module prepares a data and send it.	Data is encrypted according to the SSL RECORD protocol, by means of the cryptographic key associated to the connection, and sent to the Platform.			
2.	The Platform receives the data and reads it.	The Platform correctly decrypts data according to the SSL RECORD, by means of the cryptographic key associated to the connection and shared with the module. The, the Platform sends back an encrypted response message.			
3.	The module receives a response message by the Platform.	The module can correctly decrypt the message and check its integrity because it possesses the correct keys shared with the Platform and established by the SSL connection. Eventually the module prints the response message.			

GLOBAL TEST RESULT

☐ PASS ☐ FAIL

COMMENTS: The SSL record protocol is analyzed by logs or by inspection of the channel.

Tester in charge:	DATE:

Figura 5.10: Test di integrità e confidenzialità

Capitolo 6

Analisi delle performance

6.1 SSL vs No SSL

Sulla base della soluzione presentata e su richiesta dal team di progetto, sono state effettuate una serie di prove di perfomance per capire quanto l'approccio alla sicurezza potesse intaccare le prestazioni della piattaforma. In particolare, per le misurazioni si è fatto riferimento al RTT (round trip time), ossia il la misura del tempo impiegato da un pacchetto per viaggiare da un computer della rete ad un altro e tornare indietro. La nostra misura temporale è comprensiva ovviamente dei tempi di elaborazione sulle rispettive macchine.

6.1.1 Ambiente di test

Il test è stato effettuato su rete wireless locale (LAN), con tre macchine distinte adibite ai seguenti scopi:

- un service consumer (il connettore):
  - CPU: Intel Core i7-740QM 1.7GHz (quad core)
  - RAM: 6 GB
  - OS: Windows 7 (64bit)
- un service provider (il servizio):
  - CPU: Intel Core i3 M370 2.4GHz (dual core)
  - RAM: 4 GB
  - OS: Windows 7 (64bit)
- la Certification Authority:
  - CPU: Intel Core i7-740QM 1.7GHz (quad core)
  - RAM: 6 GB
  - OS: Ubuntu Server 12.04

I test sono stati ripetuti variando il numero di richieste (1,10,100....) verso il servizio e calcolato di volta in volta il RTT totale delle richieste. Ad esempio con 10 richieste, si calcola l'RTT di ogni richiesta e poi viene fatta la somma, ottenendo l'RTT totale; in questo modo ci aspettiamo che le curve risultanti abbiano un andamento lineare.

Sono state prese misure al variare di tre fattori:

- l'uso di SSL o meno;
- il linguaggio del connettore, Java o C++;

- la tipologia di controllo delle revoche:
  - con "revocaCRL", si intende il sistema tramite il quale il servizio scarica la CRL a intervalli temporali e, ogniqualvolta gli giunge una richiesta, fa il controllo della revoca per il certificato del richiedente su quella CRL memorizzata.
  - con "revocaHTTP", si intende il sistema tramite il quale il servizio, ogniqualvolta gli giunge una richiesta, fa il controllo della revoca per il certificato del richiedente andando ad interpellare direttamente la CA con chiamata HTTP.

Calcolando le misure al variare di questi fattori, le abbiamo infine confrontate tra loro sotto due aspetti: a parità di linguaggio, a parità di sicurezza.

6.1.2 Risultati

A parità di linguaggio, con l'esclusione della stima con revoca HTTP poiché troppo poco performante e che inoltre rendeva poco leggibile il grafico, abbiamo ottenuto i primi due grafici (Figura 6.1 e Figura 6.2) mentre a parità di sicurezza abbiamo ottenuto i restanti grafici (Figura 6.3, Figura 6.4 e Figura 6.5).

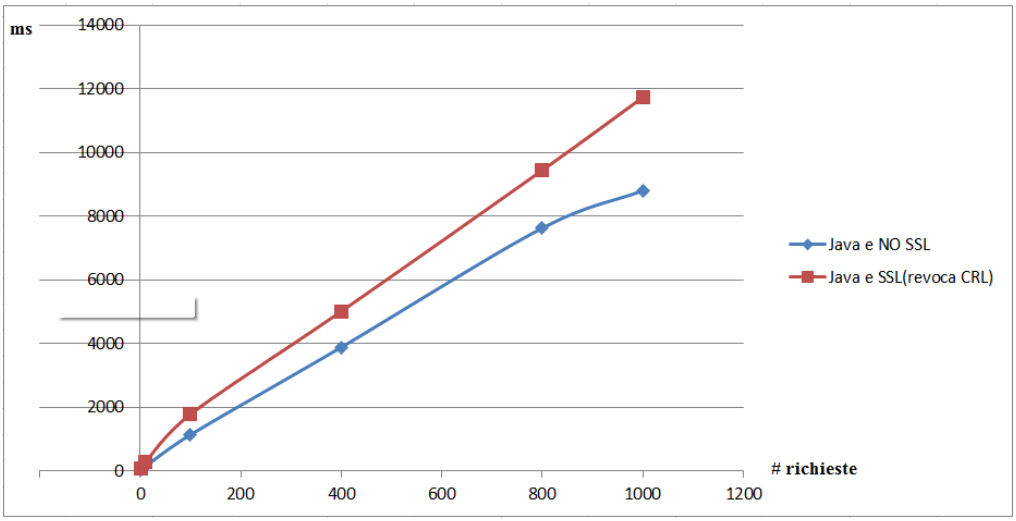


Figura 6.1: RTT totale in millisecondi al variare del numero di richieste per il connettore Java, con SSL e senza SSL

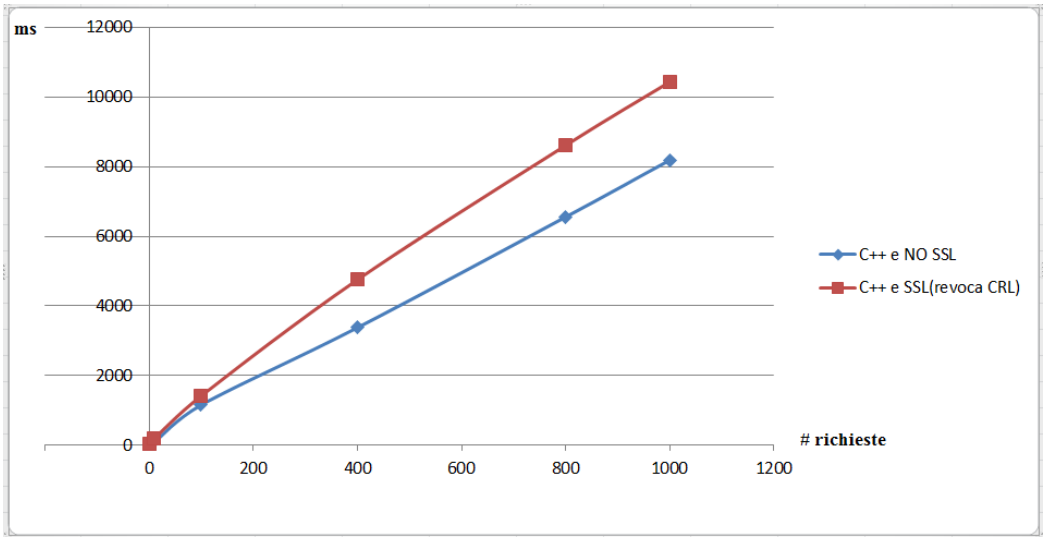


Figura 6.2: RTT totale in millisecondi al variare del numero di richieste per il connettore C++, con SSL e senza SSL

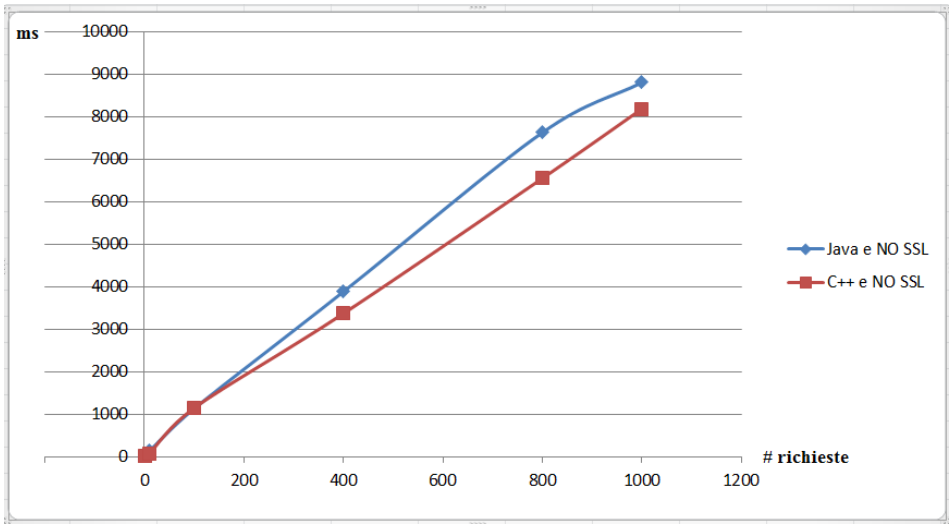


Figura 6.3: RTT totale in millisecondi al variare del numero di richieste per i due connettori, in assenza di sicurezza

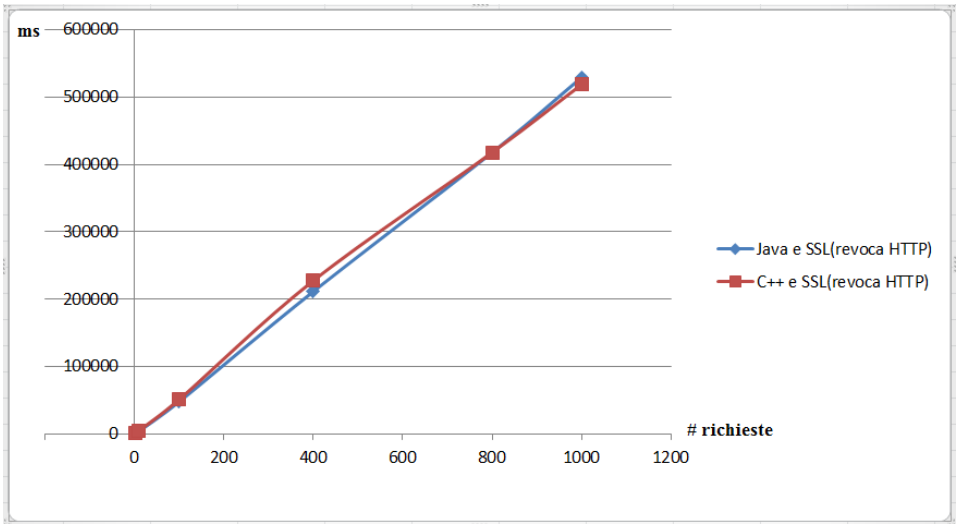


Figura 6.4: RTT totale in millisecondi al variare del numero di richieste per i due connettori, con l'uso di SSL e controllo delle revoche con richieste continue

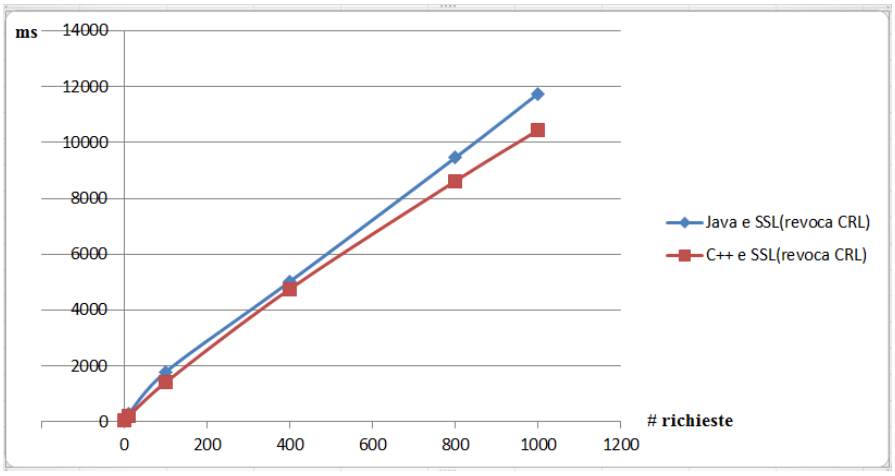


Figura 6.5: RTT totale in millisecondi al variare del numero di richieste per i due connettori,con l'uso di SSL e controllo delle revoche con polling della CRL ad intervalli temporali

6.1.3 Considerazioni finali

Innanzitutto possiamo notare che le curve hanno tutte andamento lineare come ci si aspettava avendo preso come riferimento i tempi totali delle singole richieste. Dai grafici inoltre si può osservare che:

- per il caso a parità di sicurezza: gli andamenti sono pressoché simili, con una leggera superiorità nei tempi da parte del connettore sviluppato in C++; questo era abbastanza atteso data la differente tipologia di linguaggi, uno compilato e uno semi-interpretato.

- per il caso a parità di linguaggio: questo è il risultato che più interessa poiché era atteso chiaramente un certo overhead qualora fosse stato aggiunto un livello di sicurezza alla piattaforma; ma possiamo osservare che i tempi di comunicazione per l'approccio con SSL e con-

trollo delle revoche con CRL, al variare del numero di richieste, poco si discostano dal caso senza sicurezza. Questo risultato è accettabile e quindi un buon compromesso in funzione dei benefici apportati alle comunicazioni. Infine, per quanto riguarda il caso del controllo delle revoche con interrogazioni continue alla CA, si è rivelato decisamente poco performante e dunque è stato scartato.

Capitolo 7

Conclusioni e Sviluppi futuri

Il supporto alla sicurezza offerto dal metodo proposto asseconda le richieste progettuali e garantisce i requisiti desiderati. In ottica presente è la soluzione più adatta, garantendo buone performance per le esigenze delle Piattaforma ma bisognerà constatare come si comporterà una volta terminato il progetto e quindi una volta che tutte le componenti saranno agganciate al sistema.

In ottica futura, nulla vieta di pensare a nuove soluzioni laddove ad esempio si vorrà distribuire anche l'ESB. Dal punto di vista della sicurezza invece, una possibile strada per rendere ancora più efficiente la Piattaforma potrebbe essere quella di testare il controllo delle revoche, piuttosto che con le CRL, con un nuovo protocollo emergente OCSP, tra l'altro supportato dalla Certification Authority selezionata.

Inoltre sarebbe interessante, utilizzando il prototipo di servizio e i connettori sviluppati, testare il grado di sicurezza delle configurazioni del protocollo, utilizzando ad esempio tool come Nessus Scanner per identificare eventuali vulnerabilità note, di cui il sistema potrebbe essere affetto.

Bibliografia

[1] N. Josuttis, *SOA in Practice*. O'reilly, 2007.

[2] OASIS, "Reference architecture foundation for service oriented architecture version 1.0," Dicembre 2012. [Online]. Available: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.html

[3] G. Gerla, "Java modulare con apache karaf(parte ii)," Marzo 2014. [Online]. Available: http://www2.mokabyte.it/cms/article.run?articleId=O6G-X69-TEY-7UM_7f000001_10364476_bb8a22f0#

[4] J. Edstrom and J. Goodyear, *Instant OSGi Starter*. Packt Publishing Ltd, 2013.

[5] C. Costantini, "Java modulare con apache karaf(parte i)," Marzo 2014. [Online]. Available: http://www2.mokabyte.it/cms/article.run?articleId=O6G-X69-TEY-7UM_7f000001_10364476_bb8a22f0#

[6] G. Hohpe and B. Woolf, *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[7] C. Ibsen and J. Anstey, *Camel in action*. Manning Publications Co., 2010.

[8] M. Warecki, "Understanding enterprise integration patterns," Settembre 2012. [Online]. Available: http://architects.dzone.com/articles/enterprise-integration

[9] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1," 2000.

[10] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web services description language (wsdl) 1.1," 2001.

[11] M. van Steenderen, "Universal description, discovery and integration," *SA Journal of Information Management*, vol. 2, no. 4, 2000.

[12] R. Kanneganti and P. Chodavarapu, *SOA security*. Dreamtech Press, 2008.

[13] T. Dierks, "The transport layer security (tls) protocol version 1.2," 2008.

[14] E. Rescorla, *SSL and TLS: designing and building secure systems*. Addison-Wesley Reading, 2001, vol. 1.

[15] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama *et al.*, "Web services security (ws-security)," *Specification, Microsoft Corporation*, 2002.

[16] D. Eastlake, J. Reagle, and D. Solo, "Extensible markup language) xml-signature syntax and processing," Technical report, RFC 3275, March 2002. Available from: ftp://ftp. isi. edu/in-notes/rfc3275. txt, Tech. Rep., 2002.

[17] X. E. Syntax, "Processing. w3c recommendation 10 december 2002," *Authors: Takeshi Imamura, Blair Dillaway, Ed Simon*, 2001.

[18] D. Venturi, *Crittografia nel Paese delle Meraviglie.* Springer, 2012.

[19] D. Solo, R. Housley, and W. Ford, "Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile," 2002.

[20] D. Sosnoski, "Java web services: The high cost of (ws-) security," *IBM deve-loperWorks*, 2009.